# Signal/collect12: processing large graphs in seconds

Stutz, Philip ; Strebel, Daniel ; Bernstein, Abraham

Abstract: Both researchers and industry are confronted with the need to process increasingly large amounts of data, much of which has a natural graph representation. Some use MapReduce for scalable processing, but this abstraction is not designed for graphs and has shortcomings when it comes to both iterative and asynchronous processing, which are particularly important for graph algorithms. This paper presents the Signal/Collect programming model for scalable synchronous and asynchronous graph processing. We show that this abstraction can capture the essence of many algorithms on graphs in a concise and elegant way by giving Signal/Collect adaptations of algorithms that solve tasks as varied as clustering, inferencing, ranking, classification, constraint optimisation, and even query processing. Furthermore, we built and evaluated a parallel and distributed framework that executes algorithms in our programming model. We empirically show that our framework efficiently and scalably parallelises and distributes algorithms that are expressed in the programming model. We also show that asynchronicity can speed up execution times. Our framework can compute a PageRank on a large (>1.4 billion vertices, >6.6 billion edges) real-world graph in 112 seconds on eight machines, which is competitive with other graph processing approaches.

DOI: https://doi.org/10.3233/SW-150176

Semantic Web 0 (0) 1
IOS Press



# Signal/Collect

*Processing Large Graphs in Seconds*

**Editor(s):** Andreas Hotho, Universität Würzburg, Germany
**Solicited review(s):** Jacopo Urbani, VU Amsterdam, The Netherlands; Michael Granitzer, University of Passau, Germany; Sang-Goo Lee, Seoul National University, South Korea

Philip Stutz [a], Daniel Strebel [a], Abraham Bernstein [a]
[a] *University of Zurich*

**Abstract.**
Both researchers and industry are confronted with the need to process increasingly large amounts of data, much of which has a natural graph representation. Some use MapReduce for scalable processing, but this abstraction is not designed for graphs and has shortcomings when it comes to both iterative and asynchronous processing, which are particularly important for graph algorithms.

This paper presents the Signal/Collect programming model for scalable synchronous and asynchronous graph processing. We show that this abstraction can capture the essence of many algorithms on graphs in a concise and elegant way by giving Signal/-Collect adaptations of algorithms that solve tasks as varied as clustering, inferencing, ranking, classification, constraint optimisation, and even query processing. Furthermore, we built and evaluated a parallel and distributed framework that executes algorithms in our programming model. We empirically show that our framework efficiently and scalably parallelises and distributes algorithms that are expressed in the programming model. We also show that asynchronicity can speed up execution times.

Our framework can compute a PageRank on a large (>1.4 billion vertices, >6.6 billion edges) real-world graph in 112 seconds on eight machines, which is competitive with other graph processing approaches.

Keywords: Distributed Computing, Scalability, Programming Abstractions, Programming Models, Graph Processing

## 1. Introduction

The Web (including the Semantic Web) is full of graphs. Hyperlinks and RDF triples, tweets and social network relationships, citation and trust networks, ratings and reviews – almost every activity on the web is most naturally represented as a graph. Graphs are versatile data structures and can be considered a generalisation of other important data structures such as lists and trees. In addition, many structures—be it physical such as transportation networks, social such as friendship networks, or virtual such as computer networks—have natural graph representations.

Graphs were at the core of the Semantic Web since its beginning. RDF, the core standard of the Semantic Web, represents a directed labeled graph. Hence, all processing of Semantic Web data includes at least some graph processing. Initially, many systems tried to use traditional processing approaches. Triple stores, for example, tried to leverage research in relational databases to gain scalability. The most significant speed-gains, however, came from taking into account the idiosyncrasies of storing graphs [44,59]. Another example would be the advantages gained in non-

This paper is a significant extension of [55]. Speficically, it contains a more detailed description of the programming model, describes a larger selection of algorithm adaptations, contains more extensive evaluations, particularly of the distributed version of the underlying framework.

This work is supported by the Hasler Foundation under grant number 11072.

standard reasoning through the use of graphs: Learning on the Semantic Web was initially based on using traditional propositional learning methods. It was the use of statistical relational learning methods that leveraged the graph-nature of the data that allowed combining statistical and logical reasoning [30]. This combination lead to significant gains.

Coupled with the ever expanding amounts of computation and captured data [23], this means that researchers and industry are presented with the opportunity to do increasingly complex processing of growing amounts of graph structured data.

In theory, one could write a scalable program from the ground up for each graph algorithm in order to achieve the maximum amount of parallelism. In practice, however, this requires a lot of effort and is in many cases unnecessary, because many graph algorithms such as PageRank can be decomposed into small iterated computations that each operate on a vertex and its local neighbourhood (or messages from the neighbours). If we can design programming models to express this decomposition and execute the partial computations in parallel on scalable infrastructure, then we can hope to achieve scalability without having to build custom-tailored solutions.

MapReduce is the most popular scalable programming model [10], but has shortcomings with regard to iterated processing [4,13,62,63] and requires clever mappings to support graph algorithms [9,35]. Such limitations of more general programming models have motivated specialised approaches to graph processing [29,41]. Most of these approaches follow the bulk-synchronous parallel (BSP) model [57], where a parallel algorithm is structured as a sequence of computation and communication steps that are separated by global synchronisations. The rigid pattern of bulk operations and synchronisations does not allow for flexible scheduling strategies.

To address the limitations of BSP, researchers have designed programming models for graph processing that are asynchronous [37], allow hierarchical partial synchronisations [32], make synchronisation optional [55], or try to emulate the properties of an asynchronous computation within a synchronous model [58].

Our proposed solution is a vertex-centric programming model and associated implementation for scalable graph processing. It is designed for scaling on the commodity cluster architecture. The core idea lies in the realisation that many graph algorithms can be decomposed into two operations on a vertex: (1) signaling along edges to inform neighbours about changes in vertex state and (2) collecting the received signals to update the vertex state. Given the two core elements we call our model SIGNAL/COLLECT. The programming model supports both synchronous and asynchronous scheduling of the signal and collect operations.

Such an approach has the advantage that it can be seen like a graph extension of the actors programming approach [22]. Developers can focus on specifying the communication (i.e., graph structure) and the signaling/collecting behavior without worrying about the specifics of resource allocation. Since SIGNAL/COLLECT allows mutiple types of vertices to coexist in a graph the result is a powerful framework for developing graph-centric systems. A foundation especially suitable for Semantic Web applications, as we showed in our development of TripleRush [56] – a high-performance, in-memory triple store implemented with three different vertex types.

We extend our previous work [55] on SIGNAL/COLLECT with a more detailed description of the programming model, a larger selection of algorithm adaptations, a distributed version of the underlying framework, and with more extensive evaluations on larger graphs. Given the above, our contributions are as follows:

- *We designed an expressive programming model for parallel and distributed computations on graphs.*
  We demonstrate its expressiveness by giving implementations of algorithms from categories as varied as clustering, ranking, classification, constraint optimisation, and query processing. The programming model is also modular and composable: Different vertices and edges can be combined in the same graph and reused in different algorithms. Additionally the model supports asynchronous scheduling, dataflow computations, dynamic graph modifications, incremental recomputations, aggregation operations, and automated termination detection. Note that especially the dynamic graph modifications are central for Web of Data applications as they require the seamless integration of ex-ante unknown data.

- *We evaluated a framework that implements the model.*
  The framework efficiently and scalably parallelises and distributes algorithms expressed in the

programming model. We empirically show that our framework scales to multiple cores, with increasing dataset size, and in a distributed setting. We evaluated real-world scalability by computing PageRank on the Yahoo! AltaVista web-graph.[1] The computation on the large (>1.4 billion vertices, >6.6 billion edges) graph took slightly less than two minutes using eight commodity machines, which is competitive with PowerGraph.

– *We illustrate the impact of asynchronous algorithm executions.*
SIGNAL/COLLECT supports both synchronous and asynchronous algorithm executions. We compare the difference in running times between the asynchronous and synchronous execution mode for different algorithms and problems of varying hardness.

In Section 2 we motivate the programming model and describe the basic approach. We then introduce the SIGNAL/COLLECT programming model in Section 3 and describe our implementation of the model in Section 4. In Section 5 we evaluate both the programming model and the implementation. We continue with a description of related approaches to scalable graph processing and compare them to our approach in Section 6. In Section 7 we examine the limitations of our evaluation and provide an outlook on future work. We finish by sharing our conclusions in Section 8.

## 2. The SIGNAL/COLLECT Approach: an Intuition

SIGNAL/COLLECT can be understood as a vertex-centric graph processing abstraction akin to Pregel [41]. Another way of looking at it is as an extension of the asynchronous actor model [22], where each vertex represents an actor and edges represent the communication structure between actors. The graph abstraction allows for the composition and evolution of complex systems, by adding and removing vertices and edges. To illustrate this intuition we provide two examples: RDFS subclass inferencing and the computation of the single source shortest path.

**RDFS Subclass Inferencing** Consider a graph with RDFS classes as vertices and edges from superclasses to subclasses (i.e., `rdfs:subClassOf` triples). Every vertex has a set of superclasses as state, which initially only contains itself. Now all the superclasses send their own states as signals to their subclasses, which collect those signals by setting their own new state to the union of the old state and all signals received. It is easy to imagine how these steps, when repeatedly executed, iteratively compute the transitive closure of the `rdfs:subClassOf` relationship in the vertex states.

**Single Source Shortest Path** Consider a graph with vertices that represent locations and edges that represent paths between locations. We would like to determine the shortest path from a special location `S` to all the other locations in the graph.

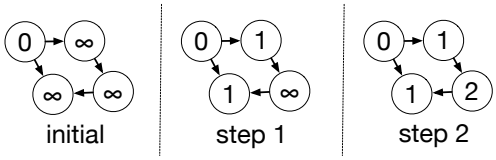



Fig. 1. States of a synchronous single-source shortest path computation with four vertices

Every location starts out with its state set to the length of the shortest currently known path from `S`. That means, initially, the state of `S` is set to 0 and the states of all the other locations are set to infinity (see Figure 1). In a first step, all edges signal the state of their source location plus the path length (represented by edge weight, in the example above all paths have length 1) to their respective target location. The target locations collect these signals by setting their new state to the lowest signal received (as long as this is smaller than their state). In a second step, the same signal/collect operations get executed using the updated vertex states. By repeating the above steps these operations iteratively compute the lengths of the shortest paths from `S` to all the other locations in the graph.

In the next section we refine this abstraction to a programming model that allows to concisely express algorithms similar to these examples.

## 3. The SIGNAL/COLLECT Programming Model

In the SIGNAL/COLLECT programming model all computations are executed on a graph, where the ver-

[1] Yahoo! Academic Relations, Yahoo! AltaVista Web Page Hyperlink Connectivity Graph, `http://webscope.sandbox.yahoo.com/catalog.php?datatype=g`

tices and edges both have associated data and computation. The vertices interact by the means of signals that flow along the edges. Vertices collect the signals and perform some computation on them employing, possibly, some vertex-state, and then signal their neighbours in the graph.

### 3.1. *Basic* SIGNAL/COLLECT *Graph Structure*

The basis for any SIGNAL/COLLECT computation is the graph

$$G = (V, E),$$

where $V$ is the set of vertices and $E$ the set of edges in $G$. During execution signals (essentially messages containing algorithm specific data items) are sent between vertices along edges.

More specifically, every *vertex* `v` $\in V$ is a computational unit that can maintain some state and has an associated *collect* function that updates the state based on the prior state and received signals. Vertices have at least the following attributes:

- $\mathbf{v}.\mathbf{id}$, a unique id.
- $\mathbf{v}.\mathbf{state}$, the current vertex state which represents computational intermediate results. The algorithm definition needs to specify an initial state.
- $\mathbf{v}.\mathbf{outgoingEdges}$, a list of all edges `e` $\in E$ with `e.source = v`.
- $\mathbf{v}.\mathbf{signalMap}$, a map with the ids of vertices as keys and signals as values. Every key represents the id of a neighbouring vertex and its value represents the most recently received signal from that neighbour. We use the alias `v.signals` to refer to the list of values in `v.signalMap`.
- $\mathbf{v}.\mathbf{uncollectedSignals}$, a list of signals that arrived since the collect operation was last executed on this vertex.

An *edge* `e` $\in E$ is also a computational unit directionally connecting two vertices that has an associated *signal* function that specifies what information is extracted from its source vertex and signaled to its target vertex. Hence, every *edge* `e` $\in E$ has the following attributes:

- $\mathbf{e}.\mathbf{source}$, a reference to the source vertex
- $\mathbf{e}.\mathbf{target}$, a reference to the target vertex

In the most general model *signals* are messages containing algorithm specific data items. The computational model makes no assumption about the structure of signals beyond that they are computed by signal functions of the edges along which they are transmitted and processed by the collect function of the target vertex.

In a practical implementation, vertices, edges, and signals will most probably be implemented as objects. We outline such an implementation in Section 4. For this reason we also allow for additional attributes on the vertices and edges.

**Example** Consider data about people and family relationships between them: How could one map this to the SIGNAL/COLLECT programming model? The most direct approach is to represent each person with a vertex. If the name of a person is unique, it could be used as the ID of the vertex, if not, then the name could be stored as an attribute on the vertex. The 'motherOf' relationship could then be represented as a directed edge between the vertices that represent the mother and her child.

To specify an algorithm in the SIGNAL/COLLECT programming model one needs to define the types of vertices in the compute graph with their associated `collect()` functions and the types of edges in the compute graph with their associated `signal()` functions. Note that the implementation of the `signal()` and `collect()` functions are interdependent. The `signal()` function creates a signal, which the `collect()` functions needs to be able to process. The `collect()` function in turn updates the vertex' state, which the `signal()` function needs to be able to read. These methods can both return values of arbitrary types, which means that vertex states and signals can have arbitrary types as well. To instantiate the algorithm for execution one needs to create the actual compute graph consisting of instances of the vertices, with their initial states, and the edges. Here one needs to specify which two vertex instances are connected by an edge instance. The result is an instantiation of a graph that can be executed.

We have now defined the basic structures of the programming model. In order to completely define a SIGNAL/COLLECT computation we still need to describe how to execute computations on them.

### 3.2. *The Computation Model and Extensions*

In this section we specify how both synchronous and asynchronous computations are executed in the SIG-

NAL/COLLECT programming model. Also we provide extensions to the core model.

In order to precisely describe the scheduling we will need additional operations on a vertex. These operations broadly correspond to the scheduler triggering communication (doSignal) or a state update (doCollect):

**v.doSignal**()
```
lastSignalState := state
for all (e ∈ outgoingEdges) do
  e.target.uncollectedSignals.append(
    e.signal())
  e.target.signalMap.put(
    e.source.id, e.signal())
end for
```

**v.doCollect**()
```
state := collect()
uncollectedSignals := Nil
```

The additional `lastSignalState` attribute on vertices stores the vertex state at the time of signalling, which is later used for automated convergence detection.

The `doCollect()` method updates the vertex state using the algorithm-specific `collect()` method and resets the `uncollectedSignals`. The `doSignal()` method computes the signals for all edges and relays them to the respective target vertices. With these methods we can describe a synchronous SIGNAL/COLLECT execution.

*3.2.1. Synchronous Execution*

A synchronous computation is specified in Algorithm 1. Its parameter `num_iterations` defines the number of iterations (computation steps the algorithm is going to perform).

Everything inside the inner loops is executed in parallel, with a global synchronization between the signaling and collecting phases. This parallel programming model is more generally referred to as Bulk Synchronous Parallel (BSP) [57].

**Algorithm 1** Synchronous execution
```
for i ← 1..num_iterations do
  for all v ∈ V parallel do
    v.doSignal()
  end for
  for all v ∈ V parallel do
    v.doCollect()
  end for
end for
```

This specification allows the efficient execution of algorithms, where every vertex is equally involved in all steps of the computation. However, in many algorithms only a subset of the vertices is involved in each part of the computation. In the next subsection we introduce scoring in order to be able to define a computational model that enables us to guide the computation and give priority to more "important" operations.

*3.2.2. Extension: Score-Guided Execution*

In order to enable the scoring (or prioritizing) of `doSignal()` and `doCollect()` operations, we need to extend the core structures of the SIGNAL/COLLECT programming model and define two additional methods on all vertices $v \in V$:

**v.scoreSignal**() : **Double**
: is a method that calculates a number that reflects how important it is for this vertex to signal. Schedulers assume that the result of this method only changes when the `v.state` changes. by default it returns 0 if `state == lastSignalState` and 1 otherwise. This captures the intuition that it is desirable to inform the neighbours iff the state has changed since they were informed last. Note that `lastSignalState` is initially uninitialised, which ensures that by default a vertex signals at least once at the start.

**v.scoreCollect**() : **Double**
: is a method that calculates a number that reflects how important it is for this vertex to collect. A scheduler can assume that the result of this method only changes when `uncollectedSignals` changes. By default it returns `uncollectedSignals.size()`. This captures the intuition that the more new information is available, the more important it is to update the state.

The defaults can be changed to methods that capture the algorithm-specific notion of "importance" more ac-

curately, but these methods should not modify the vertex.

Note: We have the scoring functions return doubles instead of just booleans, in order to enable the scheduler to make more informed decisions. Two examples where this can be beneficial: One can implement priority scheduling, where operations with the highest scores are executed first, or the scheduling could depend on some threshold (for example for PageRank), which allows the scheduler to decide at what level of precision a computation is considered converged.

Now that we have extended the basic model with scoring, we specify a score-guided synchronous execution of a SIGNAL/COLLECT computation in Algorithm 2. There are three parameters that influence when the algorithm stops: `s_threshold` and `c_threshold`, which set a minimum level of "importance" for the `doSignal()` and `doCollect()` operations to get scheduled and `max_iter`, which limits the number of iterations. The algorithm stops when either the maximum number of iterations is reached or all scores are below their thresholds. In the second case we say that the algorithm has converged. Note that the thresholds are used to configure an algorithm to skip signal/collect operations and setting them too high can lead to imprecise or wrong results.

**Algorithm 2** Score-guided synchronous execution

```
done := false
iter := 0
while (iter < max_iter and !done) do
  done := true
  iter := iter +1
  for all v ∈ V parallel do
    if (v.scoreSignal() > s_threshold) then
      done := false
      v.doSignal()
    end if
  end for
  for all v ∈ V parallel do
    if (v.scoreCollect() > c_threshold)
    then
      done := false
      v.doCollect()
    end if
  end for
end while
```

*3.2.3. Extension: Asynchronous Execution*

We referred to the first scheduling algorithm as synchronous because it guarantees that all vertices are in the same "loop" at the same time. With a synchronous scheduler it can never happen that one vertex executes a signal operation while another vertex is executing a collect operation, because the switch from one phase to the other is globally synchronised.

Asynchronous scheduling removes this constraint: Every vertex can be scheduled out of order and no global ordering is enforced. This means that a scheduler can, for example, propagate information faster by signaling right after collecting. It also simplifies the implementation of the scheduler in a distributed setting, as there is no need for global synchronisation.

**Algorithm 3** Score-guided asynchronous execution

```
ops := 0
while (ops < max_ops and ∃v ∈ V(
      v.scoreSignal() > s_threshold or
      v.scoreCollect() > c_threshold)) do
  S := choose subset of V
  for all v ∈ S parallel do
    Randomly call either v.doSignal() or
    v.doCollect() iff respective threshold is
    reached; increment ops if an operation was
    executed.
  end for
end while
```

Algorithm 3 shows a score-guided asynchronous execution. Again, three parameters influence when the asynchronous algorithm stops: `s_threshold` and `c_threshold` have the same function as in the synchronous case; `max_ops`, in contrast, limits the number of operations executed instead of the number of iterations. This guarantees that an asynchronous execution either stops because the maximum number of operations is exceeded or because it converged. The purpose of Algorithm 3 is not to be executed directly, but to specify the constraints that an implementation of the model has to satisfy during an asynchronous execution. This freedom of allowing an arbitrary execution order is useful, because if an algorithm no longer has to maintain the execution order of operations, then one is able to use different scheduling strategies for those operations.

We refer to the scheduler that we most often use as the "eager" asynchronous scheduler: In order to speed up information propagation this scheduler calls

`doSignal` on a vertex immediately after `doCollect`, if the signal score is larger than the threshold.

#### *3.2.4. Distinction: Data-Graph vs. Data-Flow Vertex*

When using the synchronous scheduler without scoring (Algorithm 1), then the collect function processes the signals that were sent along all edges during the last signaling step. When we introduce scoring, not all edges might signal during every step. There is a similar issue with asynchronous scheduling: While no signal might be forwarded along some edges, other edges might have forwarded multiple signals. For this reason we distinguish two categories of vertices that differ in the way they collect signals:

**Data-Graph Vertex** A data-graph vertex is most similar to the behaviour of a vertex in the basic execution mode: It processes `v.signals`, all the values in the signal map. This means that only the most recent signal along each edge is collected. If multiple signals were received since signals were last collected, then all but the most recent one are never collected. If no signal was sent along an edge, but there was a previous signal along that edge, then this older signal is collected for that edge. This vertex type is suitable for most graph algorithms.

**Data-Flow Vertex** A data-flow vertex is more similar to an actor. It collects all signals in `v.uncollectedSignals`, which means that it collects all signals that were received since the last collect operation. This vertex type is suitable for asynchronous data-flow processing and some graph algorithms, such as Delta-PageRank (see 5.1).

### *3.3. Extension: Graph Modifications and Incremental Recomputation*

SIGNAL/COLLECT supports graph modifications during a computation. They can be triggered externally or from inside the `doSignal()` and `doCollect()` methods. This means that vertices and edges can dynamically modify the very graph they are a part of. Modifications include adding/removing/modifying vertices/edges or sending signals from outside the graph along virtual edges (i.e., sending a message to a vertex with a known id without adding an explicit edge to the graph).

When an edge is added or removed, a scheduler has to update `scoreSignal()` and `scoreCollect()` of the respective vertex in order to check if the modification should trigger a recomputation. This is enough to support incremental recomputation for many algorithms and modifications. For some algorithms, however, additional recomputations are also required for vertices when *incoming* edges change. The more powerful incremental recomputation scheme described in [5] could be adapted to cover these cases, but would require an additional extension to track changes of incoming edges.

Modifications are always applied in per-source FIFO order, which means that all the modifications that are triggered by the same source are applied in the same order in which they were triggered. There are no guarantees regarding the global ordering of modifications.

## 4. The Signal/Collect Framework — An Implementation

The SIGNAL/COLLECT framework provides a parallel and distributed execution platform for algorithms specified according to the SIGNAL/COLLECT programming model. In this section we explain some interesting aspects of our implementation.

The framework is implemented in Scala, a language that supports both object-oriented and functional programming features and runs on the Java Virtual Machine. We released the framework under the permissive Apache License 2.0[2] and develop the source code publicly, inviting external contributions.

### *4.1. Architecture*

The framework can both parallelise computations on multiple processor cores, as well as distribute computations over a cluster. Internally, the system uses the Akka[3] distributed actor framework for message passing.

The different system components such as the coordinator and workers are implemented as actors. The coordinator bootstraps the workers and takes care of global concerns such as convergence detection and preventing messaging overload. Each worker is responsible for storing a partition of the vertices.

The scheduling of operations and message passing is done within workers. Figure 2 shows that each node

[2] `https://github.com/uzh/signal-collect` and `http://www.signalcollect.com`

[3] `http://akka.io/`

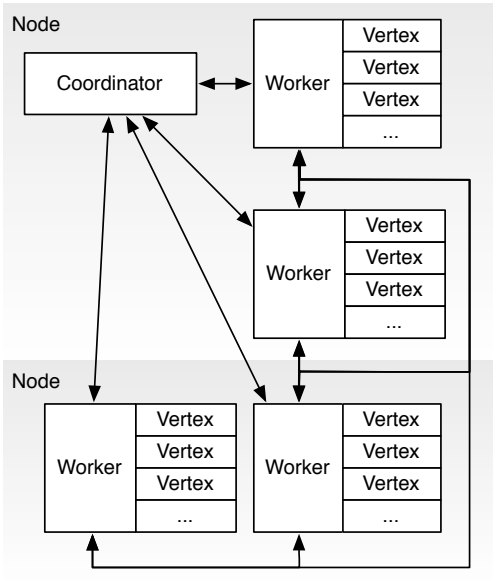



Fig. 2. Coordinator and worker actors, edges represent communication paths. Workers store the vertices.

hosts a number of workers and each worker is responsible for a partition of the vertices. Workers communicate directly with each other and with the coordinator. Workers have a pluggable scheduler that handles the delivery of signals to vertices and the ordering of signal/collect operations.

Vertices are retrieved from and stored in a pluggable storage module, by default implemented by an in-memory hash map, especially optimised for storing vertices and for efficiently supporting the operations required by the workers.

A vertex stores its outgoing edges, but neither the vertex nor its outgoing edges have access to the target vertices of the edges. In order to efficiently support parallel and distributed execution, modifications to target vertices from the model are translated to messages that are passed via a message bus.

Every worker and the coordinator have a pluggable message bus that takes care of sending signals and translating graph modifications to messages. SIGNAL/COLLECT also has implementations that support features such as bulk-messaging or signal combiners.

### *4.2. Aggregation Operations*

The framework also supports MapReduce-style aggregations over all vertices: The map function is applied to all vertices in the graph. The reduce function aggregates the mapped values in arbitrary order. Aggregation operations are used to compute global results or to define termination conditions over the entire graph.

### *4.3. Graph Partitioning and Loading*

Workers have ids from 0 ascending and by default the graph is partitioned by using a hash function on the vertex ids. This is similar to how graphs are partitioned in most other graph processing frameworks. For large graphs it usually leads to similar numbers of vertices per partition, but also to a large number of edges between partitions. To improve on this one could adopt some of the optimisations used in the Graph Processing System (GPS) [50]. Because computing a balanced graph partitioning with minimal capacity between partitions is a hard problem itself [1], this would mainly improve performance in cases where algorithms are run on the same graph repeatedly, for long-running algorithms, or when messaging bandwidth is the main bottleneck (also see discussion of limitations in Section 7).

The default storage implementation keeps the vertices in memory for fast read and write access. Extensions for secondary storage can be implemented [54]. Graph loading can be done sequentially from a coordinator actor or preferably in parallel, where multiple workers load parts of the graph at the same time. Specific partitions can be assigned to be loaded by particular workers. This can be used to have each worker load its own partition, which increases the locality of the loading.

### *4.4. Flexible Tradeoffs*

Our framework has defaults that work for a broad range of algorithms, but are not the most efficient solution for most of them. These default implementations can be replaced allowing a graph algorithm developer to choose the trade-off between implementation effort and resulting performance.

An example of a tradeoff is the propagation latency vs. messaging overhead: While sending each signal as soon as possible leads to a low latency and can perform well in local computations, sending each signal by itself will cause a lot of overhead in a distributed setting. Our implementation allows to plug in a custom bulk scheduler and bulk message bus implementation to choose a trade-off that suits the use case (throughput vs. latency).

In spite of this flexibility, our SIGNAL/COLLECT implementation is optimised for sparse graphs, due to the internally used adjacency list structure. For very dense graphs alternative representations, for example as a compressed matrix, might perform better.

### *4.5. Convergence and Termination*

The framework has to decide when an algorithm execution ends. It is not in general possible to say which algorithms can converge: The SIGNAL/COLLECT functions allow for arbitrary code execution and is hence subject to the halting problem.

For this reason the question of convergence and termination are algorithm-specific. The framework terminates as soon as an algorithm has converged, according to the score-guided execution definitions in Section 3. The framework also allows for other termination conditions in case convergence was not reached before another condition: One can give a step limit for synchronous computations and a time limit for both synchronous and asynchronous computations. The framework also supports convergence criteria based on global aggregations that are executed in step intervals for synchronous computations or in terms of time intervals for asynchronous computations.

There is also a continuous asynchronous mode where the framework keeps running and executes operations incrementally as they are triggered by modifications and signals. This mode is used by TripleRush [56] and raises the question of how to detect when a query has finished executing, given that the execution can branch many times and the number of signals/results is usually not known a priori. We solved this by implementing per-query convergence detection on top of SIGNAL/COLLECT: Every query carries a number of tickets with it and when the execution branches the tickets are split up among all branches. The vertex that does the final result reporting knows how many tickets to expect in total and can report success when all the initially sent out tickets have arrived. We described this case in more detail, because it displays that convergence detection is algorithm-specific and that there is a lot of flexibility when it comes to determining convergence criteria: it is even possible to build a custom convergence detection on top of the SIGNAL/COLLECT model.

## 5. Evaluation

In this section we evaluate the programming model, the scalability of our implementation, and the impact of asynchronous scheduling. The different contributions require different research methods: We evaluate the programming model by adapting important algorithms in a few lines of code. In addition to the expressiveness, we also show that our implementation is able to transparently scale algorithms by empirically measuring the speedup when running algorithms while varying the number of worker threads and cluster nodes. Finally, we compare the impact of asynchronous scheduling versus synchrononous scheduling on different graphs and algorithms.

### *5.1. Programming Model*

One of our main contributions is the simple, compact, yet expressive programming model. Whilst simplicity of a program is difficult to judge objectively, compactness and expressiveness are easier to show.

We demonstrate the expressiveness by giving adaptations of ten algorithms from categories as varied as clustering, ranking, classification, constraint optimisation and query processing. We show an actual implementation of the PageRank algorithm in Figure 21 in the Appendix. As the example illustrates in comparison to the pseudocode in Figure 3, the translation to executable code is straightforward.

Most algorithms are presented in a simplified version, more advanced versions of many of the examples are available online.[4] To enhance readability and facilitate the comparison of different algorithms, each algorithm is structured in a table representing the three core elements of a computation: The *initial state* represents the state of the vertices when they get added to the graph. The *collect* method uses the vertex state, the appropriate signals for the vertex type, and other vertex attributes/methods to compute a new vertex state. The *signal* method uses attributes/methods defined on the source vertex and edge attributes/methods to compute the signal that is sent along the edge. All described algorithms work on homogeneous graphs that use only one type of vertex/edge, which is specified in the table. Additional information and explanations for complex functions are provided in the algorithm descriptions.

[4] `https://github.com/uzh/signal-collect/tree/master/src/test/scala/com/signalcollect/examples`

Unless stated otherwise, the described algorithms use the default `scoreSignal()` and `scoreCollect()` implementations for automated convergence detection.

*PageRank* This graph algorithm computes the importance of a vertex based on the link structure of a graph [46]. The vertex state represents the current rank of a vertex (Figure 3). The signals represent the rank transferred from the source vertex to the target vertex. The vertex state is initialised with the `baseRank = 0.15` and the damping factor is usually set to `0.85`. There is an example of how this pseudocode directly maps to an actual algorithm implementation in the Appendix (see Figure 21).

Convergence: If one looks how the initial rank from a single vertex spreads, then one notices that it decays with the damping factor on every hop and that it eventually tapers off to zero. The computation on all vertices can be seen as many such single-source PageRank computations (sometimes referred to as personalized PageRank) that are overlaid and will, hence, also converge. In practice the convergence to zero can take many iterations, especially when there are cycles present. For this reason we usually set the scoreSignal function to return the delta between the current state of the vertex and the last signaled state (often referred to as the residual). This allows to conveniently set the desired level of precision by for example setting a signal threshold of 0.001. This means that a vertex will only signaling if the residual is still larger or equal to 0.001.

It is also possible to implement PageRank as a *dataflow algorithm* by signaling only the rank deltas (Figure 4). This version can be further optimised by not sending the source vertex id with the signals, which saves bandwidth.

*Single-source shortest path (SSSP)* This algorithm computes the shortest distance from one special source vertex (S) to all the other vertices in the graph. The vertex states represent the shortest currently known path from `S` to the respective vertex (Figure 5). Edge weights are used to represent distance. The signals represent the total path length of the shortest currently known path from `S` to `edge.target` that passes through `edge`.

Convergence: Vertices only signal if their distance was lowered by an incoming signal. Given that the distances can never be smaller than zero and that the first change of a vertex's state will set it to a finite number, the distance of a vertex can only be lowered a finite number of times, which means that the algorithm is guaranteed to eventually converge.

*Vertex Colouring* A vertex colouring problem is a special constraint optimisation problem that is solved when each vertex has an assigned colour from a set of colours and no adjacent vertices have the same colour. The following simple and inefficient algorithm solves the vertex colouring problem by initially assigning to each vertex a random colour from some arbitrary set of colours (Figure 6). Then, the vertices check if their own colour (state) is already occupied (contained in the collection of received signals). If such a conflict is encountered, they switch to a random colour except their current colour. The default `scoreSignal()` method ensures that the vertex signals again if there was a conflict. If there was no conflict, then the vertex stays with its current colour.

Algorithms such as this one can solve many optimisation problems such as scheduling or finding solutions for Sudoku puzzles. The described algorithm works with undirected edges. In SIGNAL/COLLECT these are modelled using two directed edges. The performance could be improved by using a better optimisation algorithm of which many have SIGNAL/COLLECT adaptations.[5]

Convergence: The computation keeps on going until there are no more conflicts between colours. If the number of colours available is smaller than the chromatic number of the graph, then there is no solution without conflicts, which means that this algorithm is not guaranteed to converge.

*Label Propagation* This iterative graph clustering algorithm assigns to each vertex the label that is most common in its neighbourhood [64]. Our variant is called Chinese Whispers Clustering [3] and has applications in natural language processing. The algorithm works on graphs with undirected edges which are modelled with two directed edges.

The vertex state represents the current vertex label (= cluster) and it is initialised with the vertex id (Figure 7). This means that each vertex starts in its own cluster. Then, labels are propagated to neighbours. When a vertex receives neighbours' labels, it appends its own

[5] A Distributed Stochastic Algorithm implementation in SIGNAL/COLLECT, for example, can be found at: `https://github.com/elaverman/signal-collect-dcops/blob/2e25766c04d66a6cdce4ec5a659fb0dfc45436d6/src/main/scala/com/signalcollect/approx/flood/DSA.scala`

| | |
|---|---|
| `initialState` | `baseRank` |
| `collect()` | `return baseRank + dampingFactor * sum(signals)` |
| `signal()` | `return source.state * edge.weight / sum(edgeWeights(source))` |

Fig. 3. PageRank (data-graph)

| | |
|---|---|
| `initialState` | `baseRank` |
| `collect()` | `return oldState + dampingFactor * sum(uncollectedSignals)` |
| `signal()` | `stateDelta = source.state - source.signaledState`<br>`return stateDelta * edge.weight / sum(edgeWeights(source))` |

Fig. 4. Delta PageRank (data-flow).

| | |
|---|---|
| `initialState` | `if (isSource) 0 else infinity` |
| `collect()` | `return min(oldState, min(signals))` |
| `signal()` | `return source.state + edge.weight` |

Fig. 5. Single-source shortest path (data-graph/data-flow).

| | |
|---|---|
| `initialState` | `randomColour` |
| `collect()` | `if (contains(signals, oldState))`<br>`  return randomColorExcept(oldState)`<br>`else`<br>`  return oldState` |
| `signal()` | `return source.state` |

Fig. 6. Vertex colouring (data-graph).

label to the collection of labels signalled by the neighbours. It then updates its own label to the most frequent label in that extended collection. Ties are broken arbitrarily.

The convergence depends on the mostFrequentValue function: According to [3] the algorithm does not converge if that function does not break ties in a consistent way, but that only a few iterations are needed until almost-convergence.

*Relational Classifier* Relational classification can be considered a generalisation of label propagation. The presented classifier (Figure 8) is a variation of the probabilistic relational-neighbour classifier described by Macskassy and Provost [39,40]. The algorithm works on graphs with undirected edges which are modelled with two directed edges. `ProbDist` represents a probability distribution over different classifications. Each vertex starts with an initial probability distribution over the classes, which can be uniform or can reflect the observed frequencies of classes in the training set. If the class of a vertex is available as training data, then that class gets probability 1 and is not changed by the algorithm. When a vertex receives the distributions of its neighbours, then it updates its own distribution to a normalised sum of the class probability distributions of the neighbours. A collective inference scheduling can be chosen using the `scoreSignal()` implementation to determine when a vertex informs its neighbours about its label distribution. This classifier only works when edges are more likely to connect vertices that belong to the same class (homophily) and, given its supervised nature, when some classes are known as training data [39]. The algorithm described here can be extended with a local classifier to determine the initial state. This initial state would be added to the sum of neighbour states in the `collect` method (potentially with a higher weight). In this case, it is no longer necessary for some classes to be known. The homophily con-

| | |
|---|---|
| `initialState` | `id` |
| `collect()` | `return mostFrequentValue(append(oldState, signals))` |
| `signal()` | `return source.state` |

Fig. 7. Label propagation (data-graph).

straint could be dropped by using a more advanced relational classification algorithm [6,15].

Convergence: According to [39] there is no guarantee of convergence, but according to [40] one can extend the algorithm with simulated annealing to ensure and control convergence.

*Conway's Game of Life (Life)* Life is played on a large checkerboard of cells, where each cell can be in one of two states (dead/alive) [14]. The game progresses in turns and each turn the state of a cell is updated based on the states of its neighbouring cells. The game is mapped to SIGNAL/COLLECT by representing each cell as a vertex, alive is represented with state 1 and dead as state 0 (Figure 9). The neighbourhood relationships between the cells are modeled with edges between neighbouring cells. A vertex counts how many of its neighbours are alive and uses this to determine its state next turn accroding to the rules of the game. Life is famous for the complex patterns that can emerge from its simple rules.

Convergence depends on the initial configuration and there are many (famous) initial configurations that will never converge.

*Threshold Models of Collective Behaviour* Granovetter [17] describes threshold models of collective behaviour to model situations in which agents have two options and the risk/payoff of each option depends on the behaviour of neighbouring agents. The risk/payoff is determined by a threshold which can be different for every actor. Threshold models allow to model the collective behaviour of a group of actors. Granovetter uses the example of rioting, but argues that such models can also be used to model innovation, rumor diffusion, disease spreading, strikes, voting, migration, educational attainment, and attendance of social events.

Such models can be mapped to SIGNAL/COLLECT by representing each agent with a vertex and connecting all agents that can observe each other's behaviour with edges. The initial state determines the default behaviour of an agent (Figure 10). In our example, an actor does not riot initially, unless it is a natural rioter, which means that the agent would riot no matter what its neighbours do. Edges inform neighbours about the behaviour of an agent and in the `collect()` method the agent analyses what fraction of its neighbours are taking the alternative decision. If the fraction of neighbouring agents that display a behaviour exceeds the individual threshold (in our example `riotingThreshold`), then the agent changes its behaviour and switches to the alternative behaviour (e.g., start to riot).

Convergence: In the described model no person will ever stop rioting, once they decide to riot. For this reason only a finite number of 'I am now rioting' messages can ever be sent. This means that at some point either everyone is rioting, or the non-rioters will never receive an additional message that could shift them towards becoming rioters. For this reason, the computation is guaranteed to converge.

*Matching Path Queries* This algorithm matches path queries, which is a typical use case for a graph processing system. The signals sent in the algorithm initially come from outside the graph along a virtual edge. The signals are path queries that specify a pattern of vertices and edges that they can match. An example for such a pattern might be: Match any path that starts with a vertex that has a "professor" property, continues along an edge that has an "advises" property and ends with an arbitrary vertex.

Once a query arrives at a vertex, its first part is matched with the vertex at which it has arrived (Figure 11). This is done with the `successfulMatchesWithVertex()` function, which returns only the queries that have successfully matched with the local vertex.

If the query is fully matched—meaning all parts of its path are bound to a vertex or edge—then this path is reported as a result (this could be done by adding it to some result attribute that is later picked up by an aggregation operation). If there is still a part of the query left that needs to be matched, then it is added to the state set of the vertex. During the signal operation all edges try to match the next part of the queries—the one potentially constraining the type of edge to follow—using their `successfulMatchesWithEdge()` functions. Queries that were successfully edge-matched are

| | |
|---|---|
| `initialState` | `if (isTrainingData) trainingData else avgProbDist` |
| `collect()` | `if (isTrainingData)`<br>`  return oldState`<br>`else`<br>`  return signals.sum.normalise` |
| `signal()` | `return source.state` |

Fig. 8. Relational classifier (data-graph).

| | |
|---|---|
| `initialState` | `if (isInitiallyAlive) 1 else 0` |
| `collect()` | `switch (sum(signals))`<br>`  case 0: return 0  // dies of loneliness`<br>`  case 1: return 0  // dies of loneliness`<br>`  case 2: return oldState // same as before`<br>`  case 3: return 1  // becomes alive if dead`<br>`  other:  return 0  // dies of overcrowding` |
| `signal()` | `return source.state` |

Fig. 9. Conway's Game of Life (data-graph).

| | |
|---|---|
| `initialState` | `if (isNaturalRioter) true else false` |
| `collect()` | `riotingNeighbours = filterTrue(signals)`<br>`rioterFraction = riotingNeighbours.size / signals.size`<br>`if (rioterFraction > riotingThreshold)`<br>`  return true`<br>`else`<br>`  return false` |
| `signal()` | `return source.state` |

Fig. 10. Threshold models of collective behaviour (data-graph).

returned by that function and signalled along the respective edges.

Matching path queries has many use cases: From simpler ones such as triangle/cycle detection, the approach could be extended to more complex tasks such as computing random walks with restarts or even matching expressive graph query languages.

Convergence: If each query only has a finite number of expected vertex/edge matches, then the execution is guaranteed to converge, because all queries will eventually either be eliminated or become fully matched.

*Artificial Neural Networks* Artificial neural networks are the result of an attempt to imitate the structure of biological neural networks and there are *"literally tens of thousands of published applications"* [49, p. 748]. Neural networks consist of nodes connected by links [49, p. 737]. The nodes are mapped to vertices in SIGNAL/COLLECT and the links are mapped to directed edges. Activations are sent as signals between edges, with the difference that they already get adjusted for edge weight in the `signal()` method of the edge (Figure 12). The activation function is mapped to the `collect()` method and updates the state that represents the unit activation. Varying inputs are sent from outside the graph as signals along virtual edges.

Convergence: Neural networks usually do not contain cycles, so if there are no more new inputs, then the remaining activations are guaranteed to finish propagating through the network at some point.

*Sketching of some additional algorithms* The "Bipartite Matching" and "Semi-Clustering" algorithms described in the Pregel paper [41] can be adapted to the SIGNAL/COLLECT model by separating the compute function into a signal part for communication and a collect part for the state update. They require access to the step number, which is not available in the default SIGNAL/COLLECT model, but can be added for example by using parallel update operations between com-

| | |
|---|---|
| initialState | emptySet |
| collect() | matched = successfulMatchesWithVertex(signals)<br>(fullyMatched, partiallyMatched) = partition(matched)<br>reportResults(fullyMatched)<br>return union(oldState - lastSignalState, partiallyMatched) |
| signal() | return successfulMatchesWithEdge(source.state) |

Fig. 11. Matching path queries (data-flow).

| | |
|---|---|
| initialState | 0 |
| collect() | return 1 / (1 + $e^{-\text{signals.sum}}$) |
| signal() | return source.state * edge.weight |

Fig. 12. Artificial neural networks (data-graph).

putation steps. An adaptation of "Loopy Belief Propagation" has been outlined in [55] and was used to do inference on Markov logic networks [51].

We have also implemented a triple store with competitive performance inside SIGNAL/COLLECT [56]. This system uses three different vertex types to model an index and to keep track of query executions.

The main benefit of adapting an algorithm or system to the SIGNAL/COLLECT model is that its execution is automatically scheduled in parallel (or even distributed, if several machines are available), which allows for scalability. In the next section we empirically evaluate the scalability of our framework when executing algorithms expressed in the programming model.

### 5.2. Scalability

In this subsection we evaluate the scalability of the SIGNAL/COLLECT framework. In order to evaluate this we empirically measure the performance of our framework on multiple algorithms while varying the available resources. More specifically, we analyse the scalability by varying (1) the number of worker threads, (2) the size of the processed graph, and (3) the number of cluster nodes.

#### 5.2.1. Multi-core (vertically/scale up)

We determined the multi-core scalability by measuring the parallel speedup when running an algorithm on the same graph, but with a varying number of worker threads.

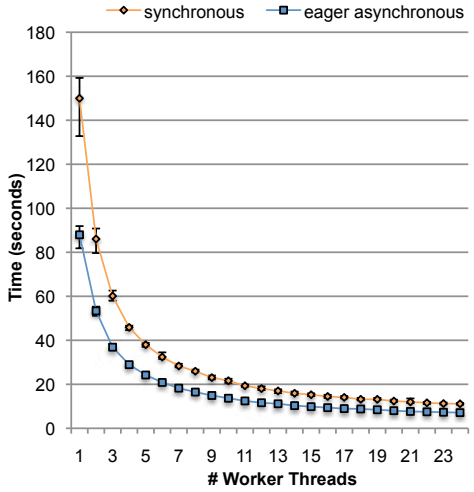
In a first benchmark we ran the SSSP and PageRank algorithms on a machine with two twelve-core AMD Opteron™ 6174 processors and 66 GB RAM. We executed these algorithms with both synchronous and "eager" asynchronous scheduling. They were run on the web graph dataset[6] with 875 713 vertices (websites) and 5 105 039 edges (hyperlinks). Each combination of algorithm and scheduler was run whilst varying between 1 and 24 worker threads (as the machine has 24 cores). Each run was executed ten times and we graphed the resulting average running time in Figures 13 (PageRank) and 14 (SSSP), where the error bars indicate min/max running times. The speedup was calculated relative to the average runtime with one worker thread. Each execution was run cold in a new JVM, because this reflects the actual usage best.

PageRank was run with a signal function that returns the delta between the previous signal state and the current state (see Figure 4). The signal threshold was set to 0.01, which determines the precision of the result. More detailed evaluation parameters can be found in the evaluation program.[7]

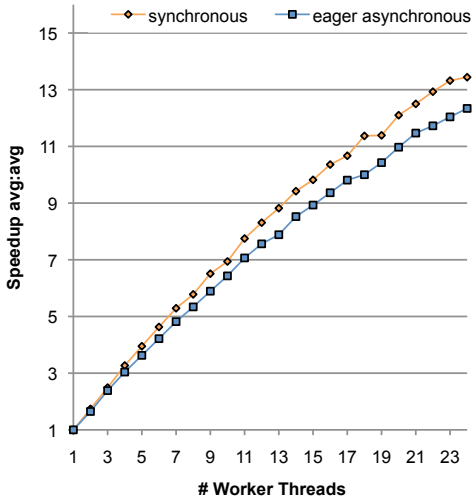
The fastest running time was 7 seconds for PageRank and 1.2 seconds for SSSP. The speedup when going from 1 core to 24 cores was 9 for SSSP and around 13 for PageRank. This shows that SIGNAL/COLLECT scales with additional cores, but that the actual factor depends on the algorithm. The achievable speedup also

[6] http://snap.stanford.edu/data/web-Google.html

[7] The evaluation program used was MulticoreScalabilityEvaluation in https://github.com/uzh/signal-collect-evaluation at revision 05057d000d. The snapshot dependencies were https://github.com/uzh/signal-collect at revision ba26e95e20 and https://github.com/uzh/signal-collect-graphs at revision 0149927e68. The results are available at https://docs.google.com/spreadsheet/ccc?key=0AiDJBXePHqCldEVwYk1lWDJpQmVRc0QtUWxLcFVXUWc.

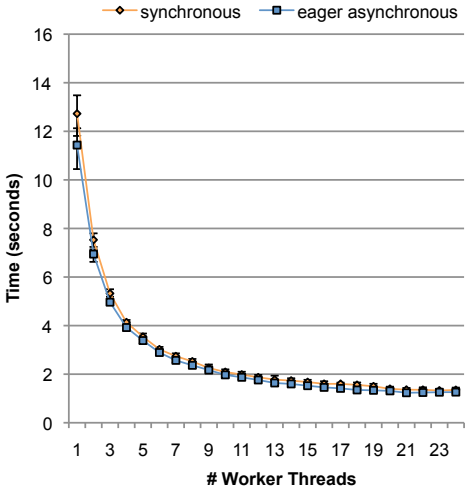



(a) Execution Times

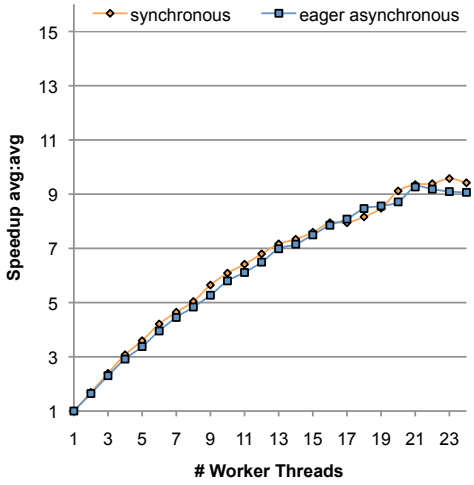



(b) Speedup

Fig. 13. Multi-Core Scalability of PageRank




(a) Execution Times




(b) Speedup

Fig. 14. Multi-Core Scalability of Single-Source Shortest Path

depends on the graph structure: Running SSSP on a chain of vertices would not allow for any parallelism with our implementation.

### *5.2.2. Data scalability*

In order to evaluate how SIGNAL/COLLECT scales with increasing graph sizes we had to determine what kind of graphs to evaluate it on. Given that we want the graphs to be as similar as possible, Kronecker graphs [34] seem like a good choice, because they preserve many properties of a graph across different scales. We used the reference implementation of the generator that is part of SNAP [2].[8]

We generate the graphs by using the fitted parameters for the Notre Dame web graph ([0.999 0.414; 0.453 0.229]) which we also got from [34]. With these parameters we generated graphs with between 20 to 26 iterations of the Kronecker product, which resulted in graphs with between 659 518 vertices and 2 652 653 edges up to 39 865 268 vertices and 224 276 985 edges, in between increasing approximately with powers of two. For 27 iterations the graph generator repeat-

[8] `https://snap.stanford.edu/snap/`

edly threw errors (the machine on which it was run had 128 GB of RAM).

The partitions in of the generated graphs are in many ways unbalanced: The number of outgoing edges from a worker partition varies by a factor of around 9 between the worker with the most outgoing edges and the one with the fewest. With regards to message sending, the distribution is even more uneven: We checked the number of sent messages between the busiest worker for one run at 20 and 26 iterations, and in both cases the busiest worker sent more than 100 times as many messages as the least busy worker.

We ran the experiment on machines with 128 GB RAM and two E5-2680 v2 processors at 2.80GHz, with 10 cores per processor. The JVM on the machines used between 300 MB (minimum on the smallest graph) and 12.5 GB of RAM (maximum on the largest graph).

Figure 15 shows the performance when running delta PageRank with a threshold of 0.01 on these synthetic graphs.[9] We ran all evaluations 10 times and plot the average execution time. We tested the running times with both a specialised signal combiner for the rank deltas and with the generally applicable bulk messaging. The plot is logarithmic in both axes and it shows that SIGNAL/COLLECT scales linearly with increasing dataset sizes. Whilst bulk messaging performs better for smaller graph sizes, the signal combiner is faster on the largest dataset.

### 5.2.3. *Distribution (horizontally/scale out)*

We determined the distributed scalability by measuring the speedup when running an algorithm on the same graph, but with a varying number of cluster nodes.

Whilst the previous benchmarks ran with a simple PageRank implementation, we ran an optimised version of the Delta PageRank algorithm on the Yahoo! AltaVista webgraph[10] with 1 413 511 390 vertices and 6 636 600 779 edges. This is one of the largest real-world graphs available for such evaluations and a realistic use case for the PageRank algorithm. In order to measure scalability we ran the algorithm ten times with each configuration on a cluster with 4, 6, 8, 10, and 12 nodes, with 24 worker threads per node (i.e., 96 workers up to 288 workers). The nodes were the same 24-core machines used in the multi-core scalability evaluation connected by a 1 gigabit ethernet switch. In all computations the coordinator actor was running on a laptop on a different network, this machine had no problem handling the load of running termination detection and flow control. The latency between coordinator and workers was less than one millisecond. Given that the heartbeat interval was 100 milliseconds, it is unlikely for the remoteness of the coordinator to have had much effect on the computation.

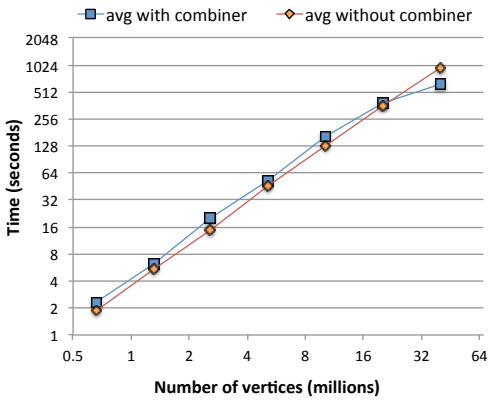



Fig. 15. Data scalability for PageRank on synthetic Kronecker graphs. Both axes have logarithmic scales.

The vertices were partitioned using hashing as described earlier, so most edges spanned different workers and nodes. The graph was loaded from the local file system of the machines and loading took between 45 seconds (fastest run with 12 nodes) and 235 seconds (slowest run with 4 nodes).

The huge number of signals required more efficient usage of bandwidth, which is why we used a bulk scheduler and bulk message bus. When scaling across more nodes and workers, this means that either each bulk signal has to contain fewer signals or that there is increased latency that would impair algorithm convergence. We chose to keep the latency constant, which has the effect of reducing the benefits of bulk signaling for runs with more nodes, but removes convergence characteristics as a confounding factor.

Another optimisation that we used was to have the vertex change the edge representation and for each edge only store the ID of the target vertex. The target

[9] The evaluation program used was PageRankEvaluation in `https://github.com/uzh/signal-collect-evaluation` at revision de1a018. The used `https://github.com/uzh/signal-collect` dependency was at revision 77da25f. The results are available at `https://docs.google.com/spreadsheet/ccc?key=0AiDJBXePHqCldFFxaFp1N0RzTDBKdzhoSHlmb3M1T0E`.

[10] Yahoo! Academic Relations, Yahoo! AltaVista Web Page Hyperlink Connectivity Graph, `http://webscope.sandbox.yahoo.com/catalog.php?datatype=g`

IDs are integers, so to further reduce the memory footprint we sorted the array of target IDs and at each position only stored the delta from the previous array entry. We then took advantage of the smaller IDs by using variable length encoding on the ID deltas. Furthermore, we collected signals right when they were delivered, which makes it unnecessary to store them inside the vertex until they are collected. These optimisations reduced the memory footprint and allowed us to successfully run the algorithm on only four machines.

Finally, delta PageRank was run with a signal function that returns the delta between the previous signal state and the current state. Every execution ran until convergence with a signal threshold of 0.01 and both the vertex state (PageRank) as well as the signals were represented as floating point numbers.

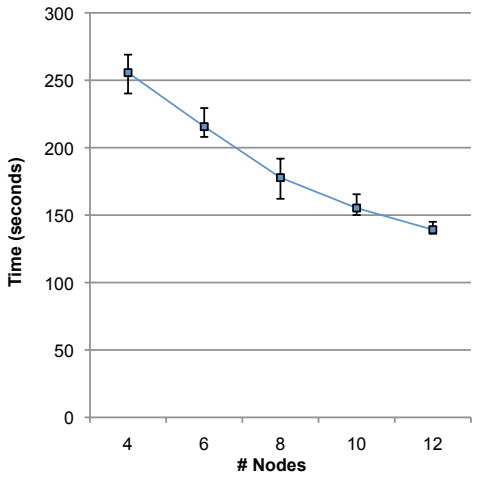
Figure 16(a) graphs the execution times. It shows that increasing the number of nodes decreases runtime. Indeed, the speedup plot in Figure 16(b) shows that for using three times more resources we get a speedup of almost two. This is decent, considering that more nodes means a larger fraction of signals are sent across nodes (over the slow network, as opposed to fast in-memory transfers) and that there is more overhead for signaling due to smaller bulk signal sizes. More detailed evaluation parameters can be found in the evaluation program.[11]

*Comparison with other reported results* We are well aware that comparing run-times between systems run on different machines is a problematic proposition at best. The main goal of the comparisons below is, therefore, to provide an intuition of the order of scalability and performance of SIGNAL/COLLECT in contrast to other systems reported on in the literature.

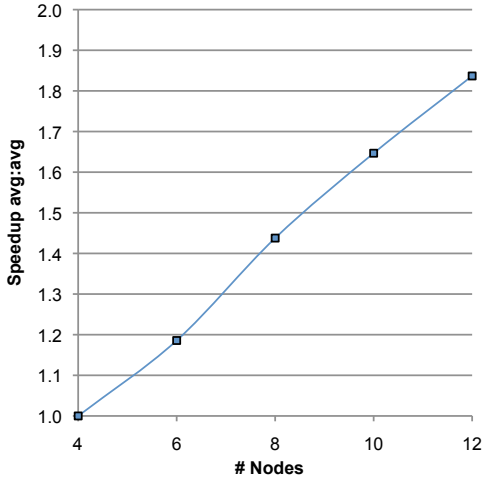
Pegasus [28] is a MapReduce-based system that ran 10 iterations of an iterative belief propagation algorithm on the same Yahoo! AltaVista webgraph using 100 machines of a Hadoop cluster. This computation took 4 hours.

GPS [50] computed 50 iterations of PageRank on a webgraph with 51 million vertices and 1.9 billion edges in *846 seconds* using a cluster of 60 Amazon EC2 nodes (4 virtual cores and 7.5GB of RAM each). Using a pre-partitioned graph reduced the computation time to 372 seconds. In their evaluations they describe that GPS runs more than an order of magnitude faster than Giraph.

GraphLab did not report any evaluations for the PageRank algorithm, which complicates comparison.




(a) Execution Times




(b) Speedup

Fig. 16. Horizontal scalability of PageRank on the Yahoo! AltaVista Web Page Hyperlink Connectivity Graph with 1 413 511 390 vertices and 6 636 600 779 edges. The data points in 16(a) show the average execution time over 10 runs and the error bars indicate the fastest and slowest runs. 16(b) plots the speedup relative to the average execution time with 4 nodes. The signal threshold used was 0.01, state and signals were represented as floats.

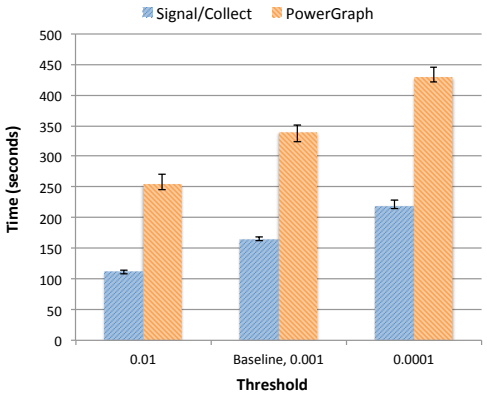
[11] The evaluation program used was DistributedWebGraphScalabilityEval in `https://github.com/uzh/signal-collect-evaluation` at revision 701e208. The snapshot dependencies were `https://github.com/uzh/signal-collect` at revision 43c3b0ffe7 and `https://github.com/uzh/signal-collect-graphs` at revision f35637c930. The results are available at `https://docs.google.com/spreadsheet/ccc?key=0AiDJBXePHqCldDF2dEJIWnliVkJ0cjBrVlVvOTBkMkE`.

The largest (pre-partitioned) graph it was evaluated on had 27 million vertices and 375 million edges. The non-partitioned ones were smaller.[12]

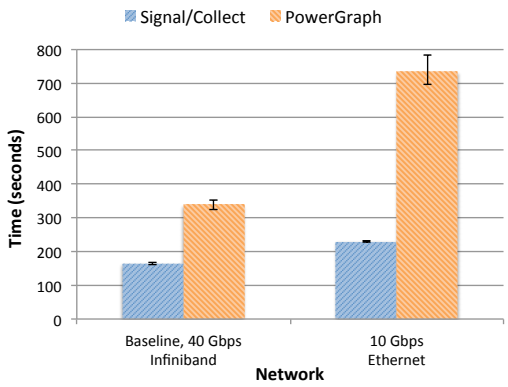
PowerGraph [16] required about 14 seconds to compute PageRank on a Twitter follower graph with *40 million vertices* and 1.5 billion edges employing a cluster of 64 Amazon EC2 nodes (8 cores and 23 GB of RAM each, connected by 10 gigabit Ethernet). They report faster times with coordinated partitioning requiring an up-front loading time of more than 200 seconds. In their evaluations PowerGraph is at least an order of magnitude faster than the other frameworks they compare against.

### 5.3. Performance comparison with PowerGraph

In order to fairly compare the performance of two systems they have to be run on the same hardware. To that end we ran PageRank on both PowerGraph and SIGNAL/COLLECT, again on the AltaVista webgraph. The experiment was run on a cluster of 8 machines, each machine having 128 GB RAM and two E5-2680 v2 processors at 2.80GHz, with 10 cores per processor. The machines were connected with both 10Gbps Ethernet and 40Gbps Infiniband. For PowerGraph we used the repository version from July 6th 2014.[13] From that version we used the existing toolkit PageRank implementation.[14] For SIGNAL/COLLECT we used a precise version of the asynchronous delta PageRank, as in the scalability evaluation, but in addition we added a message combiner that sums up ranks on the sender side.[15] Both systems use doubles internally to represent ranks and for the baseline we configured them with a tolerance of 0.001 and with enabled Infiniband. Baseline PowerGraph was run with the synchronous engine (because it is much faster than its asynchrnous engine, see 18(b)), whilst baseline delta PageRank was run with the asynchronous execution mode of SIGNAL/COLLECT. We compared the best-performing configurations for the baseline (with the exception of the threshold parameter).




(a)




(b)

Fig. 17. Comparison between SIGNAL/COLLECT and PowerGraph of the respective execution times for computing PageRank on the Yahoo! AltaVista webgraph when (a) the convergence threshold and (b) the network is varied relative to the baseline configuration.

In Figures 17 and 18 the bar displays the average running time over ten runs and the error bars indicate the performance of the fastest and slowest runs.

Figure 17(a) shows how the execution times change when the convergence threshold is varied around the baseline. We see that SIGNAL/COLLECT takes about half as long at all precision levels.

Figure 17(b) shows the comparison of the baseline configurations of PowerGraph and SIGNAL/COLLECT with one in which Infiniband is disabled. We notice that disabling Infiniband increases the running time of

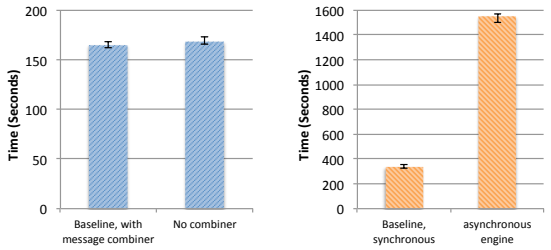
[12] GraphLab did not scale up to the Yahoo! AltaVista webgraph according to the thesis defence slides of Joseph E. Gonzalez, slide 70, `http://www.cs.cmu.edu/~jegonzal/talks/jegonzal_thesis_defense.pptx`.

[13] `https://github.com/graphlab-code/graphlab/tree/4d201c2599d51f9975617dc4a3f39f6c9d489cc4`

[14] `https://github.com/graphlab-code/graphlab/blob/4d201c2599d51f9975617dc4a3f39f6c9d489cc4/toolkits/graph_analytics/pagerank.cpp`

[15] The evaluation program used was PageRankEvaluation in `https://github.com/uzh/signal-collect-evaluation` at revision d3c7aaa. The used `https://github.com/uzh/signal-collect` dependency was at revision 17a28a2. The results are available at `https://docs.google.com/spreadsheet/ccc?key=0AiDJBXePHqCldFhUWHM2dG13emZkaUNhVGhreWZBNlE`.

SIGNAL/COLLECT by around 50%, whilst the PowerGraph execution time more than doubles. We were surprised by this, because for this graph PowerGraph reported a replication factor of 3.27, which means that for each original vertex there were on average more than 3 replicas. We expected that this would lead to a lower sensitivity to network bandwidth and latency, but the results suggest that PowerGraph is more sensitive to the quality of the network.




(a) (b)

Fig. 18. Figure 18(a) compares the baseline execution time of SIGNAL/COLLECT, which has an enabled message combiner, with the generally applicable bulk messaging.
Figure 18(b) compares the baseline execution time of PowerGraph, which uses the synchronous engine, with the asynchronous engine of PowerGraph.
Note that Figure 18(a) and Figure 18(b) do not use the same scale on the y-axis.

We also tested enabling delta caching for the PowerGraph synchronous and asynchronous engine. In both cases enabling delta caching resulted in the computation still running after several hours and we canceled the evaluation at that point.

We loaded the graphs from different formats and from a network drive, which is why the loading times are not comparable and are subject to variance due to local caching of the files. For SIGNAL/COLLECT we loaded the graph from a binary format that split vertices according to their hash code, which is why each worker only loaded local vertices. For this reason most SIGNAL/COLLECT runs loaded the graph in around 50 seconds. PowerGraph loaded the graph with the automatically determined 'grid' ingress method an adjacency list format from 1413 splits in more than 10 minutes, but this could likely be sped up by a lot if one were to also use a specialised format.

Note: PowerGraph only ran the computation on 720 242 173 vertices, because vertices without incoming or outgoing edges were discarded during the loading phase. One SIGNAL/COLLECT run and four PowerGraph runs crashed during the computation, in this case we simply restarted that run.

Memory usage: The PowerGraph allocator reported that it used around 80 GB of the heap on each machine. The memory usage per node of SIGNAL/COLLECT varied between 20 GB and 47 GB, the variance is most likely due to garbage collection timing.

We also tried out alternative configurations: Figure 18(a) compares the impact of using bulk messaging instead of the baseline signal combiner with SIGNAL/COLLECT. We see that bulk messaging is slightly slower, but that the specialised message combiner did not improve the execution time by much. Figure 18(b) compares the execution time when using the PowerGraph asynchronous engine instead of the baseline synchronous engine. We see that the asynchronous engine is more than four times slower. We found this surprising, but a PowerGraph author explains that this obvservation might be explained by the the asynchronous engine being less optimised.[16]

As always, given the complexity of such distributed frameworks, it is difficult to draw hard conclusions. Nonetheless, we feel that we can clearly state that the SIGNAL/COLLECT approach seems highly competitive when computing a popular algorithm on a web-scale graph using a conventional cluster. We can also state that SIGNAL/COLLECT's retained its competitiveness when turning off some of its more complex refinements (see Figure 18(a)) and that it was less reliant on a fast network for its performance.

### *5.4. Asynchronous Computation*

In this section discuss and evaluate some aspects of asychronous scheduling in SIGNAL/COLLECT. Depending on the scheduling strategy an asynchronous scheduler can have the following advantages:

– *Lower latency between operations*
An asynchronous scheduler is not tied to a global ordering that prescribes when information can be propagated. This flexibility can be used to reduce the latency between collecting and signaling and is especially important for use cases such as query processing, where latency is critical (see path query processing 5.1). It can also lead to fewer signal operations due to faster information propagation (see the PageRank analysis in sub-

[16] `http://forum.graphlab.com/discussion/comment/277`

section 5.4).

– *Reduction of oscillations*
Some synchronous algorithms can be prone to getting trapped in oscillation patterns, where vertices cycle through states in lockstep. Asynchronous processing can reduce such oscillations and allows some of these algorithms to converge quickly).

In order to measure how much impact the lower latency signal propagation has we ran PageRank and SSSP with both kinds of schedulers in the previously described scalability experiments. As reported in Figures 13 and 14 asynchronous scheduling is on average between 36% (with 24 workers) and 41% (with 1 worker) faster than synchronous scheduling. This is largely because of earlier signal propagation: In all cases the asynchronous version required on average 30% fewer signal operations until convergence.

Scheduling does not have the same impact on all algorithms: The single-source shortest path algorithm took approximately the same amount of time, regardless of the scheduling.

To evaluate the *impact of scheduling on oscillations* we ran a greedy algorithm to solve vertex colouring problems on the Latin Square dataset.[17] The graph is a vertex matrix with 100 columns and 100 rows, where all vertices in each column and all vertices in each row are connected (modeled by almost 2 million undirected edges in SIGNAL/COLLECT). The problem requires at least 100 colours to be solved and becomes easier to solve when more colours are available. We ran the algorithm with both synchronous and "eager asynchronous" scheduling for a varying number of available colours. The hardware used was the same as in subsection 5.2.1. In Figure 19 we show the fastest execution time of the ten runs for each number of colours. Executions were terminated after 20 minutes, even if no solution was found.[18] Each vertex was initialised with the same colour, initialising with random colours would lead to faster executions.

Figure 19 shows the executions with "eager" asynchronous scheduling found solutions much quicker than synchronous executions. For the harder problems with fewer colours there are also several cases where a synchronous scheduling fails to find a solution within the time limit, while the asynchronous scheduling found a solution within a few seconds. One explanation is that with a synchronous scheduling the vertices tend to switch states in lockstep, which has them cycle through or oscillate between conflicts ("thrashing"). The results show that for some algorithms asynchronous scheduling can be crucial for fast convergence. Other algorithms share this property: Koller and Friedman note that some asynchronous loopy belief propagation computations converge where the synchronous computations keep oscillating. They summarize in that context that [31, p. 408]: *"In practice an asynchronous message passing scheduling works significantly better than the synchronous approach."*

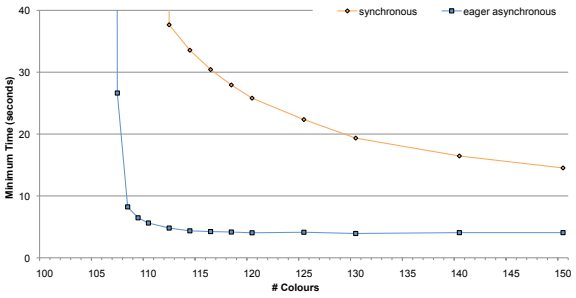



Fig. 19. Vertex colouring with a varying number of colours on a Latin Square problem with 100 * 100 vertices and almost two million directed edges.

## 6. Related Work

In this section we give an overview over the foundations of SIGNAL/COLLECT and alternative approaches to large-scale graph processing.

### 6.1. Foundations

The SIGNAL/COLLECT programming model is related to three lower-level programming models, which are all suitable for distributed and parallel computations, but lack a focus on graph computations:

[17] We used the dataset provided by CMU at `http://mat.gsia.cmu.edu/COLOR04/INSTANCES/qg.order100.col`

[18] The evaluation program used was VertexColoringSyncVsAsyncEvaluation in `https://github.com/uzh/signal-collect-evaluation` at revision f53d9897b1. The snapshot dependencies were `https://github.com/uzh/signal-collect` at revision 40d89ba1c1 and `https://github.com/uzh/signal-collect-graphs` at revision 0149927e68. The results are available at `https://docs.google.com/spreadsheet/ccc?key=0AiDJBXePHqCldEFORUhISkJVSy15Nmd6QnlqYzFKWUE`.

– *Bulk-synchronous parallel (BSP)*
  In BSP [57], a parallel computation consists of a sequence of supersteps. During each superstep, components process some assigned task and communicate with each other. There is a periodic global synchronization that ensures that all tasks of the superstep have been completed before the next superstep is started. The synchronous scheduling of SIGNAL/COLLECT is an implementation of this model.

– *Actor model*
  In the actor programming model [22], many processing components take part in a computation and operate in parallel. These components can only influence each other via asynchronous message passing. The asynchronous scheduler of SIGNAL/COLLECT was inspired by the actor model.

– *Data-flow model*
  Depending on the context, the expression "dataflow" can have different meanings. We understand it broadly as a programming model where a computation is defined as a dependency graph in which data flows along edges and vertices use their input data to compute new data that gets sent along their outgoing edges. This model can be seen as a specialisation of the actor model, where each vertex is represented by an actor and communicates along the graph structure.

When designing a programming model for graph processing, it is important to consider the different kinds of computations on graphs. There are two fundamentally different ways of thinking about computations on graphs. One way to interpret a graph is as a data structure, where data can be associated with vertices and edges. Computations may explore this structure and modify its data, potentially iteratively, until some termination or convergence criterium is reached. We refer to computations with this characteristic as *data-graph* computations. Another way to interpret a graph is as a plan that determines the flow of data along processing stages. Vertices represent processing stages for data, while edges represent the (potentially cyclic) paths along which data flows. This view encompasses the *data-flow* programming model.

With SIGNAL/COLLECT, we have designed a programming model that is suitable for both kinds of computations: In the SIGNAL/COLLECT programming model vertices are processing units akin to actors whilst edges can have associated data and may compute signals that flow to their target vertex.

Researchers with roots in disparate communities such as machine learning, biology, or the semantic web have answered the call for general programming models and frameworks specialised for scalable graph processing.

Figure 20 provides a high-level overview of distributed data processing systems that support iterated processing. It differentiates systems along their ability for synchronous versus asynchronous processing of the data on the y-axis and the kind of data abstraction they operate on (key-value pairs, sets, or tables versus graphs) on the x-axis. The category "Synchronisation Required" encompasses systems that schedule iterated computations with mandatory global barriers between iterations. The "Asynchronous Possible" category encompasses systems that are able to schedule iterated computation without such global barriers. "MR Based" is used as a category label for systems that extend the MapReduce model with support for iterated processing or graph abstractions.

We focus our discussion on programming models and systems that are geared towards processing graphs, especially ones that are capable of asynchronous processing. Specifically, we first present GraphStep and Pregel, which have inspired many other graph specialised BSP-based systems. After that, we discuss GraphLab, its extension PowerGraph, and HipG in detail, because they are vertex centric approaches that are closely related to SIGNAL/COLLECT. In the last part we give summaries of other related work.

### *6.2. GraphStep and Pregel*

*GraphStep* [12] is the programming model that is most closely related to SIGNAL/COLLECT. It combines the BSP programming model with the concept of vertices as actors and edges representing the communication structure between those actors. A computation progresses with all vertices receiving input messages, awaiting a global synchronization, performing a local update operation, and sending output messages. The last two steps broadly correspond to collecting and then signaling. A newer description of the model [11] adds a reduce phase on the incoming messages and a separate edge function where each edge reads and writes local state and then possibly sends a message to its destination vertex. The model is meant to be im-

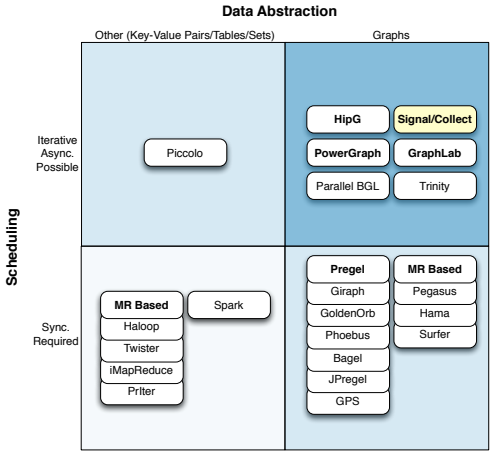



Fig. 20. **Selected Related Work:** An overview of only the high-level distributed data processing *systems* that support iterated processing. References: Piccolo[48], MapReduce[10], Hadoop[19], Twister[13], iMapReduce[62], PrIter[63], Spark[60], HipG[33], PowerGraph[16], Parallel BGL[53], GraphLab[37], Trinity[61], Pregel[41], Giraph[20], GoldenOrb[21], Phoebus[22], Bagel[23], JPregel[24], GPS[50], Pegasus[29], Hama[52], Surfer[8].

plemented in field-programmable gate array (FPGA) circuits and does not support data-flow computations, asynchronicity, or graph modifications.

*Pregel* is a framework with a similar programming model developed by Google for large-scale graph processing [41]. The framework scales to graphs with billions of vertices and edges via distribution to thousands of commodity PCs. Pregel is based on a programming model that was inspired by BSP: A computation consists of a sequence of supersteps. During a superstep, each vertex executes an algorithm-specific compute function that can modify the vertex state, modify the graph, and send messages to other vertices. Global synchronisations ensure that all compute functions of the superstep have been completed before the next superstep is started. Within a compute function, a vertex can vote to halt the computation. A computation ends when all the vertices have voted to halt. In order to reduce the number of messages that are sent between workers/machines, Pregel supports combiners that aggregate multiple messages for the same vertex into one. For the computation of global values Pregel also supports aggregation operations.

The Pregel model merges computation, communication, and termination detection into one compute function on a vertex. This function is a black box from the perspective of the framework, which requires a manual implementation of termination detection and prevents the scheduler from separately scheduling state updates and communication.

Pregel can only handle synchronous computations. In a synchronous computation one problematic operation or node can be enough to slow all computations, while in an asynchronous computation only operations on that node or the specific operation are slowed. As we discussed in our evaluation (see Section 5.4), synchronous scheduling can also lead to convergence problems due to oscillations for some algorithms.

Pregel supports graphs with one kind of vertex type sharing a single compute function.This complicates the reusability of vertex/edge-specifications and it adds complexity when implementing algorithms with multiple kinds of vertices or edges. These constraints also make it harder to compose several algorithms within the same computation.

There are extensions to the model for incremental recomputations [5] and for custom scheduling that can imitate some of the properties of an asynchronous scheduling [58].

Google has not released its implementation of Pregel, but there exist several related open source implementations such as Giraph[25], GoldenOrb[26], Phoebus[27], Bagel[28], and JPregel[29]. In addition, two Pregel-like implementations were developed with a special focus: Menthor[30] is an open source implementation associated with research into high-level control structures over computation steps [21] and the Graph Processing System (GPS) implementation supports static and dynamic graph partitioning [50]. Also noteworthy is Green-Marl [24], a domain-specific programming language for graph analysis algorithms that compiles to execution systems with a Pregel-like programming model.

### *6.3. GraphLab*

GraphLab is a programming model and framework for parallel graph algorithms [37]. The programming model is especially suitable for computations with sparse data dependencies and for asynchronous iterative computation.

[25] http://giraph.apache.org/
[26] http://goldenorbos.org/
[27] https://github.com/xslogic/phoebus
[28] https://github.com/mesos/spark/blob/master/bagel
[29] http://kowshik.github.com/JPregel/
[30] http://lcavwww.epfl.ch/~hmiller/menthor/

GraphLab is based on a data-graph model which simultaneously represents data and computational dependencies. A computation progresses by executing update functions on vertices. These functions can modify the vertex and edge data as well as data associated with neighbouring vertices in the data-graph. The model offers flexible scheduling of these update operations as well as functions to aggregate over the state of the entire data-graph. The scheduler supports different consistency guarantees, which permit the adaptation of some algorithmic correctness proofs from a sequential to a parallel setting.

In full-consistency mode, concurrent modifications to the neighbourhood of a vertex have to be prevented while an update function is executed. Assuming a random distribution of vertices over cluster nodes of a large cluster—an assumption currently true for many frameworks such as HipG (see below) and Pregel—the expected (and worst-case) scenario is that the vertex data and the data of neighbouring vertices are spread over almost as many nodes as there are vertices in the neighbourhood. The authors describe in a more recent publication [16] (see below) that executing an update function in full-consistency mode on a vertex with a sizable neighbourhood is a costly operation, because it requires a distributed lock of a large fraction of the cluster. They also mention that "distributed locking and synchronization introduces substantial latency" [36]. Furthermore, they describe that "the locking scheme used by GraphLab is unfair to high degree vertices" (see [16], Section 4.3.2).

Another scheduler ("chromatic engine") works around some of these issues in distributed computations [36]. This scheduler uses a vertex colouring to avoid the expensive locking during execution. It is equivalent to a BSP execution where at each step only vertices with the same colour are active. This scheduler requires finding a graph colouring (more constrained ones for strong consistency guarantees) and the number of processing steps and global synchronisations is multiplied by the number of colours used for the graph colouring. There seems to be a trade-off between the availability of consistency guarantees and the effort obtaining a graph colouring.

GraphLab does currently not allow graph modifications during a computation and does not support graphs with multiple vertex types, which complicates composition of algorithms and reusability of components. Lastly, GraphLab has undirected edges. Hence, algorithms that exploit directionality would have to encode it in an edge's data, complicating the framework's ability to optimise computations based on directionality.

### *6.4. PowerGraph*

PowerGraph [16] is a substantial redesign and reimplementation of GraphLab. The main difference in PowerGraph's abstraction lies in the computation's division into three phases: gather roughly corresponds to a Pregel combiner gathering and aggregating the data from neighbours, apply computes the new vertex state, and scatter updates the values of the adjacent edges. According to the execution semantics ([16], Algorithm 1, Section 4.1) the three functions are always called sequentially, without interruption, possibly complicating scheduler-based optimisations. Akin to GraphLab, PowerGraph does not allow for multiple implementations of the three functions per graph and it does not support graph modifications during computations.

To enable a more efficient implementation of the distribution PowerGraph introduces the idea of vertex cuts – essentially the replication of vertices to many machines. Whilst this reduces cross-machine communication for some algorithms and more evenly distributes the load of high-degree vertices, it introduces replication of the vertices and their associated state/-data up to a factor of 5-18 for 64 machines. The variation of the overhead factor is dependent on the partitioning strategy chosen. Smarter strategies reduce the replication factor but increase graph loading time by a factor of about 5 (when using 64 machines).

In a direct comparison of the programming models, PowerGraph has consistency guarantees and the built-in optimisations for high-degree vertices going for it. SIGNAL/COLLECT offers more flexibility with regard to the efficient edge representation, and can even route messages to another vertex if there is no explicitly stored edge between them. SIGNAL/COLLECT also has built-in support for using different vertex, edge and message types inside the same graph, whilst the same requires a custom mapping and is potentially inefficient for PowerGraph.

PowerGraph's lack of support for modifications during a computation or the efficiency of the asynchronous implementation are most likely engineering related and not fundamental properties of the approach.

### 6.5. HipG

HipG is a distributed framework that facilitates high-level programming of parallel graph algorithms by expressing them as a hierarchy of distributed computations [32,33].

As in the Pregel model, code is executed on a vertex. But while in Pregel messages are sent to other vertices, a HipG vertex can conceptually directly execute functions on neighbouring vertices (the framework translates those function calls to asynchronous messages). HipG supports synchronisers which are coordinators for function executions that have the option to block until all executed functions have completed. This feature can also be used to aggregate global values. A synchroniser can spawn additional synchronisers to create hierarchical computations. This is especially useful for divide and conquer algorithms on graphs.

While it is possible to write a compute function for a vertex that handles thousands of received messages at once, there is no obvious way of combining functions, which means that they all have to be executed. This could be problematic if one wants to implement an iterated computation, because it would require for a function to spawn as many new functions as there are neighbours, potentially leading to an exponential growth of functions in the system. One solution is to use a global synchroniser that repeatedly executes functions on all vertices (using a "visit" flag and only propagating onwards if the flag is not set yet) and has barriers (synchronisations) between those executions (indeed, this is how the PageRank example is implemented in the example code provided with the system[31]) – an implementation of a BSP-scheduler with HipG primitives.

### 6.6. Other Related Work

The Parallel BGL[32] is a generic C++ library of parallel and distributed graph algorithms and data structures [18,19,38]. One of the main design goals of this system (and also of the ParGraph[33] system) was to allow for sequential BGL[34] algorithms to be "lifted" to parallel programs whilst minimising the required changes. It has support for a special process group that delivers messages immediately instead of waiting for a BSP step synchronisation, but this feature is not explored in any depth.

Najork et al. [43] evaluated three different platforms and programming models on large graphs. The focus of the evaluation is to find the trade-offs between the platforms for various algorithm. The evaluation did not include vertex-centric models. Members of the same lab are also working on the Trinity graph engine, a distributed key-value store with optimisations for vertex-centric graph processing, such as bundling of messages, graph partitioning and low latency processing [61]. Trinity also supports asynchronous processing and while the technology and features of the framework are impressive, it is not publicly available and the report gives little information about the properties of the programming model and about how algorithms are expressed.

There are several systems for large-scale graph computations implemented on top of MapReduce or by generalising the MapReduce model. Most of these systems have limited support for iterated computations and do not support asynchronicity [8,20,29,52]. PrIter is a modified version of Hadoop MapReduce that supports executing processing steps only on a subset of items with priorities above a threshold [63]. Kajdanowicz et al. [27] compared the efficiency of a MapReduce-based system with a BSP-based system for processing large graphs and conclude that BSP can outperform MapReduce by up to an order of magnitude.

Piccolo and Spark are distributed processing platforms that use table-/set-based abstractions, support iteration and can serve as the foundation of more specialized graph processing frameworks [48,60]. One such extension is the aforementioned Bagel which is built on Spark.

Also noteworthy are distributed data-flow engines such as Sawzall [47], Dryad/DryadLINQ [25,26], Pig [45], and Ciel/Skywriting [42]. Computations on graphs require a custom mapping to the respective data-flow language model. Some of the languages allow to express iterated processing, but the underlying systems are not optimised for doing this efficiently on graph structured data.

There are more graph processing libraries that focus on specific algorithms, but did not offer a detailed enough explanation of a more general programming model. Also we did not cover frameworks that focus on specific aspects of scaling algorithms on architectures

[31] http://www.few.vu.nl/~e.krepska/HipG/

[32] Parallel Boost Graph Library: http://osl.iu.edu/research/pbgl/

[33] http://pargraph.sourceforge.net/

[34] http://www.boost.org/doc/libs/1_46_1/libs/graph/doc/index.html

such as supercomputers or GPUs, because the scalability challenges are different.

## 7. Limitations and Future Work

As we have seen in previous sections, SIGNAL/COLLECT is a scalable programming model and framework for parallel and distributed graph algorithms. Whilst our expressiveness evaluation is limited by the number of algorithms shown, their wide variability indicates some generalizability of our claim of simplicity and expressiveness. The generalizability of our scalability evaluations is also limited by the number of algorithms tested. But again, we believe that the range of algorithms is typical for such evaluations and provides a strong indication for SIGNAL/COLLECT's scalability.

In addition, our evaluation does raise some very interesting questions: Could we improve performance with a better graph partitioning scheme? How does SIGNAL/COLLECT fare with graphs containing vertices with disproportional numbers of in- and out-degree vertices? How could Signal/Collect recover from failures? And, what would the impact of a prioritising scheduler be on run-time performance? We discuss these questions in the remainder of this section.

Due to the default partitioning scheme the vast majority of signals in the distributed version are sent over the network. This is inefficient, but it could be improved without modifying the programming model: A domain optimised hash function could be used, for example one that maps websites from the same domain to the same worker or cluster node. This should improve locality of signaling. It would be interesting to see to what degree such a scheme would suffer from imbalanced loads for different domains.

We did not encounter any problems due to high in/out-degree vertices so far. Whilst Pregel-style combiners address the problem of *high in-degree vertices*, the problem of *high out-degree vertices* could be addressed by modifying a graph: High out-degree vertices could create child vertices that each inherit a share of the outgoing edges. All state changes and further edge additions/removals are forwarded to the child vertices. High out-degree child vertices could recursively use the same scheme. A more efficient and more limited alternative is to parallelise the signaling on a vertex to "smear" the signaling workload across a cluster node instead of having the entire load on one worker.

All our experiments where run on either web-style real-world graphs or the synthetic Kronecker graphs. These graphs have an uneven degree distribution [34]. Consequently, one might argue that our findings may have a limited gerenalizability. As discussed above, however, we believe that SIGNAL/COLLECT could be adapted to process high-degree vertices. We do acknowledge, that the memory overhead of representing graphs as collections of edges and vertices may rise above matrix-based representations in highly connect graphs that would result in non-sparse matrices. Note however, that such graphs seem uncommon on the (semantic) web, social networks, and many naturally occurring phenomena, as these have been found to follow a power law degree distribution [34].

SIGNAL/COLLECT currently only supports very primitive checkpointing and no error recovery. All the jobs that we have run so far were very short-lived and we never encountered hardware failures. Hence, for us it would make more sense to simply restart a failed job. With the long-running jobs and use cases such as query processing error recovery would become more important. The distributed snapshot algorithm [7] would probably be a good candidate for addressing this issue, because it could run without interrupting algorithm execution.

Finally, it would be interesting to experiment with a prioritising scheduler. Such a scheduler might have benefits for use cases in which computing and sending signals is very expensive relative to other tasks. Otherwise, the overhead of prioritising operations may not pay off.

## 8. Conclusions

Both researchers and industry are confronted with the need to process increasingly large amounts of data, much of which has a natural graph representation. In order to address the need to run algorithms on increasingly large graphs we have designed a programming model that is both simple and expressive. We showed its expressiveness by designing adaptations of many interesting algorithms to the programming model and the simplicity by being able to express these algorithms with just a few lines of code.

We built an open source framework that can parallelise and distribute the execution of algorithms formulated in the model. We empirically evaluated the scalability of the framework across different graph structures and algorithms and have shown that the frame-

work scales with additional resources. The framework offers great efficiency and performance on a cluster architecture, which was shown by loading the huge Yahoo! AltaVista webgraph and computing high-quality PageRanks for its vertices in just 3 minutes on a dozen machines.

With SIGNAL/COLLECT we have created a programming abstraction that allows programmers to run algorithms quickly on large graphs without worrying about the specifics of how parallel and distributed processing resources are allocated. We believe that marrying actors with a graph abstraction should be taken beyond simple graph processing and that this is an effective approach to building dynamic and complex systems that operate on large data sets.

## Acknowledgements

We thank the Hasler foundation for funding our research and Yahoo! for giving us access to the AltaVista Web Page Hyperlink Connectivity Graph. We thank Mihaela Verman, Lorenz Fischer, and Patrick Minder for the many interesting discussions, for feedback, and proofreading, as well as William Cohen for input on earlier versions of the project. We thank Francisco de Freitas for designing the first prototype of an Akka-based distributed version of the SIGNAL/COLLECT framework. We thank the reviewers Jacopo Urbani, Michael Granitzer, and Sang-Goo Lee for their feedback and suggestions for improvements.

## Appendix

In order to show how the framework is used in practice, we show the source code of an executable algorithm in the SIGNAL/COLLECT framework in Figure 21.[35]

[35]Source code available at `https://github.com/uzh/signal-collect-evaluation/blob/master/src/main/scala/com/signalcollect/evaluation/algorithms/paper/PageRank.scala`.

```
import com.signalcollect._

class PageRankVertex(
  id: Any, initialState: Double)
  extends DataGraphVertex(id, initialState) {
  type Signal = Double
  def collect = 0.15 + 0.85 * signals.sum
}

class PageRankEdge(targetId: Any)
  extends DefaultEdge(targetId) {
  type Source = PageRankVertex
  def signal = source.state / source.edgeCount
}

object PageRankExample extends App {
  val graph = GraphBuilder.
    withStorageFactory(
      factory.storage.JavaMapStorage).
      build
  graph.addVertex(new PageRankVertex(1, 0.15))
  graph.addVertex(new PageRankVertex(2, 0.15))
  graph.addVertex(new PageRankVertex(3, 0.15))
  graph.addEdge(1, new PageRankEdge(2))
  graph.addEdge(2, new PageRankEdge(1))
  graph.addEdge(2, new PageRankEdge(3))
  graph.addEdge(3, new PageRankEdge(2))
  graph.execute(
    ExecutionConfiguration.
      withSignalThreshold(0.001))
  val top2 = graph.aggregate(
    TopKFinder[Double](k = 2))
  top2.foreach(println(_))
  graph.shutdown
}
```

Fig. 21. Executable Scala code of a PageRank algorithm definition, sequential graph building, a local execution, and an aggregation operation. Note that the actual PageRank code only encompasses the upper two class definitions. The `PageRankExample` object builds a tiny graph, executes the computation, runs an aggregation to find the two top ranked vertices, prints them, and then shuts the system down. Both the graph building and the execution are highly configurable: In this example an alternative storage implementation is used and the signal threshold is modified, both for illustration. The implementation could be simplified by using the defaults.

## References

[1] ANDREEV, K. AND RÄCKE, H. 2004. Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. SPAA '04. ACM, New York, NY, USA, 120–124.

[2] BADER, D. A. AND MADDURI, K. 2008. Snap, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks. In *22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, IEEE Press, Miami, FL, 1–12.

[3] BIEMANN, C. 2006. Chinese whispers: an efficient graph clustering algorithm and its application to natural language process-

ing problems. In *Proceedings of the First Workshop on Graph Based Methods for Natural Language Processing*. TextGraphs-1. Association for Computational Linguistics, Stroudsburg, PA, USA, 73–80.

[4] BU, Y., HOWE, B., BALAZINSKA, M., AND ERNST, M. D. 2010. Haloop: efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment 3*, 285–296.

[5] CAI, Z., LOGOTHETIS, D., AND SIGANOS, G. 2012. Facilitating real-time graph mining. In *Proceedings of the fourth international workshop on Cloud data management*. CloudDB '12. ACM, New York, NY, USA, 1–8.

[6] CHAKRABARTI, S., DOM, B., AND INDYK, P. 1998. Enhanced hypertext categorization using hyperlinks. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. SIGMOD '98. ACM, New York, NY, USA, 307–318.

[7] CHANDY, K. M. AND LAMPORT, L. 1985. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst. 3,* 1, 63–75.

[8] CHEN, R., WENG, X., HE, B., AND YANG, M. 2010. Large graph processing in the cloud. In *SIGMOD Conference'10*. ACM Press, New York, NY, USA, 1123–1126.

[9] COHEN, J. 2009. Graph twiddling in a mapreduce world. *Computing in Science Engineering 11,* 4, 29 –41.

[10] DEAN, J. AND GHEMAWAT, S. 2004. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. USENIX Association, Berkeley, CA, USA, 10–10.

[11] DELORIMIER, M. 2013. Graph parallel actor language — a programming language for parallel graph algorithms. Ph.D. thesis, California Institute of Technology. http://resolver.caltech.edu/CaltechTHESIS:08192012-145253489.

[12] DELORIMIER, M., KAPRE, N., MEHTA, N., RIZZO, D., ESLICK, I., RUBIN, R., URIBE, T. E., KNIGHT, T. F., AND DEHON, A. 2006. Graphstep: A system architecture for sparse-graph algorithms. In *In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines. IEEE*. IEEE Computer Society, Napa, CA, 143 – 151.

[13] EKANAYAKE, J., LI, H., ZHANG, B., GUNARATHNE, T., BAE, S.-H., QIU, J., AND FOX, G. 2010. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, S. Hariri and K. Keahey, Eds. ACM, New York, NY, 810–818.

[14] GARDNER, M. 1970. Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American 223,* 4, 120–123.

[15] GETOOR, L., SEGAL, E., TASKAR, B., AND KOLLER, D. 2001. Probabilistic models of text and link structure for hypertext classification. In *In IJCAI Workshop on Text Learning: Beyond Supervision*. International Joint Conferences on Artificial Intelligence, Inc., http://linqs.cs.umd.edu/basilic/web/Publications/2001/getoor:ijcaiws01/ijcaiws.pdf.

[16] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. 2012. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. OSDI'12. USENIX Association, Berkeley, CA, USA, 17–30.

[17] GRANOVETTER, M. 1978. Threshold Models of Collective Behavior. *American Journal of Sociology 83,* 6, 1420–1443.

[18] GREGOR, D. AND LUMSDAINE, A. 2005a. Lifting sequential graph algorithms for distributed-memory parallel computation. *ACM SIGPLAN Notices - Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications 40*, 423–437.

[19] GREGOR, D. AND LUMSDAINE, A. 2005b. The parallel bgl: A generic library for distributed graph computations. In *In Parallel Object-Oriented Scientific Computing (POOSC)*. Glasgow, UK.

[20] GU, Y., LU, L., GROSSMAN, R., AND YOO, A. 2010. Processing massive sized graphs using sector/sphere. In *2010 IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS)*. IEEE Press, New Orleans, LA, 1 –10.

[21] HALLER, P. AND MILLER, H. 2011. Parallelizing machine learning– functionally: A framework and abstractions for parallel graph processing. In *2nd Annual Scala Workshop 2011*.

[22] HEWITT, C., BISHOP, P., AND STEIGER, R. 1973. A universal modular actor formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245.

[23] HILBERT, M. AND LÓPEZ, P. 2011. The world's technological capacity to store, communicate, and compute information. *Science 332,* 6025, 60–65.

[24] HONG, S., CHAFI, H., SEDLAR, E., AND OLUKOTUN, K. 2012. Green-marl: a dsl for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII. ACM, New York, NY, USA, 349–362.

[25] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. ACM, New York, NY, USA, 59–72.

[26] ISARD, M. AND YU, Y. 2009. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the 35th SIGMOD international conference on Management of data*. SIGMOD '09. ACM, New York, NY, USA, 987–994.

[27] KAJDANOWICZ, T., INDYK, W., KAZIENKO, P., AND KUKUL, J. 2012. Comparison of the efficiency of mapreduce and bulk synchronous parallel approaches to large network processing. In *2012 IEEE 12th International Conference on Data Mining Workshops (ICDMW)*. IEEE, IEEE Press, Vancouver, Canada, 218 –225.

[28] KANG, U., CHAU, D., AND FALOUTSOS, C. 2010. Inference of beliefs on billion-scale graphs. In *The 2nd Workshop on Large-scale Data Mining: Theory and Applications*. ACM, ACm press, Washington, DC.

[29] KANG, U., TSOURAKAKIS, C. E., AND FALOUTSOS, C. 2009. Pegasus: A peta-scale graph mining system. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (ICDM)*. IEEE Computer Society, Los Alamitos, CA, USA, 229–238.

[30] KIEFER, C., BERNSTEIN, A., AND LOCHER, A. 2008. Adding data mining support to sparql via statistical relational learning methods. In *The Semantic Web: Research and Applications*, S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis, Eds. Lecture Notes in Computer Science Series, vol. 5021. Springer Berlin Heidelberg, Teneriffe, Spain, 478–492.

[31] KOLLER, D. AND FRIEDMAN, N. 2009. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, Cambridge,

MA.

[32] KREPSKA, E., KIELMANN, T., FOKKINK, W., AND BAL, H. 2011a. A high-level framework for distributed processing of large-scale graphs. In *Proceedings of the 12th international conference on Distributed computing and networking*. ICDCN'11. Springer-Verlag, Berlin, Heidelberg, 155–166.

[33] KREPSKA, E., KIELMANN, T., FOKKINK, W., AND BAL, H. 2011b. Hipg: parallel processing of large-scale graphs. *ACM SIGOPS Operating Systems Review 45,* 2, 3–13.

[34] LESKOVEC, J., CHAKRABARTI, D., KLEINBERG, J., FALOUTSOS, C., AND GHAHRAMANI, Z. 2010. Kronecker graphs: An approach to modeling networks. *The Journal of Machine Learning Research 11*, 985–1042.

[35] LIN, J. AND SCHATZ, M. 2010. Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*. MLG '10. ACM, New York, NY, USA, 78–85.

[36] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., AND GUESTRIN, C. 2011. Graphlab: A distributed framework for machine learning in the cloud. CMU Tech Report 1107.0922, Carnegie Mellon University, http://arxiv.org/abs/1107.0922.

[37] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. M. 2010. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*. AUAI Press, Catalina Island, California.

[38] LUMSDAINE, A., GREGOR, D., HENDRICKSON, B., AND BERRY, J. W. 2007. Challenges in parallel graph processing. *Parallel Processing Letters 17,* 5, 5–20.

[39] MACSKASSY, S. A. AND PROVOST, F. 2003. A simple relational classifier. In *Proceedings of the Second Workshop on Multi-Relational Data Mining (MRDM-2003) at KDD-2003*. ACM Press, Washington, DC, 64–76.

[40] MACSKASSY, S. A. AND PROVOST, F. 2007. Classification in networked data: A toolkit and a univariate case study. *J. Mach. Learn. Res. 8*, 935–983.

[41] MALEWICZ, G., AUSTERN, M. H., BIK, A. J. C., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, A. K. Elmagarmid and D. Agrawal, Eds. ACM, New York, NY, 135–146.

[42] MURRAY, D. G., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. 2011. Ciel: a universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*. NSDI'11. USENIX Association, Berkeley, CA, USA, 9–9.

[43] NAJORK, M., FETTERLY, D., HALVERSON, A., KENTHAPADI, K., AND GOLLAPUDI, S. 2012. Of hammers and nails: an empirical comparison of three paradigms for processing large graphs. In *Proceedings of the fifth ACM international conference on Web search and data mining*. WSDM '12. ACM, New York, NY, USA, 103–112.

[44] NEUMANN, T. AND WEIKUM, G. 2010. x-rdf-3x: fast querying, high update rates, and consistency for rdf databases. *Proceedings of the VLDB Endowment 3,* 1-2, 256–263.

[45] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. 2008. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. SIGMOD '08. ACM, New York, NY, USA, 1099–1110.

[46] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. 1998. The PageRank citation ranking: Bringing order to the Web. Tech. rep., Stanford Digital Library Technologies Project.

[47] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. 2005. Interpreting the data: Parallel analysis with sawzall. *Sci. Program. 13,* 4, 277–298.

[48] POWER, R. AND LI, J. 2010. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. OSDI'10. USENIX Association, Berkeley, CA, USA, 1–14.

[49] RUSSELL, S. J. AND NORVIG, P. 2003. *Artificial Intelligence: a modern approach* 2nd international edition Ed. Prentice Hall, Upper Saddle River, New Jersey.

[50] SALIHOGLU, S. AND WIDOM, J. 2012. Gps: A graph processing system. Tech. rep., Stanford, http://infolab.stanford.edu/gps/publications/tech-report.pdf.

[51] SCHURGAST, S. 2010. Markov logic inference on signal/collect. M.S. thesis, University of Zurich, Department of Informatics.

[52] SEO, S., YOON, E. J., KIM, J., JIN, S., KIM, J.-S., AND MAENG, S. 2010. Hama: An efficient matrix computation with the mapreduce framework. *Cloud Computing Technology and Science, IEEE International Conference on 0*, 721–726.

[53] SIEK, J., LEE, L.-Q., AND LUMSDAINE, A. 2002. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, Indianapolis, Indiana.

[54] STREBEL, D. 2011. Making signal/collect scale. Bsc thesis, University of Zurich.

[55] STUTZ, P., BERNSTEIN, A., AND COHEN, W. W. 2010. Signal/Collect: Graph Algorithms for the (Semantic) Web. In *International Semantic Web Conference (ISWC) 2010*, P. P.-S. et al., Ed. Vol. LNCS 6496. Springer, Heidelberg, Shanghai, China, pp. 764–780.

[56] STUTZ, P., VERMAN, M., FISCHER, L., AND BERNSTEIN, A. 2013. Triplerush: A fast and scalable triple store. In *9th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2013)*. CEUR-WS, Sydney, Australia, 50–65.

[57] VALIANT, L. G. 1990. A bridging model for parallel computation. *Commun. ACM 33,* 8, 103–111.

[58] WANG, G., XIE, W., DEMERS, A., AND GEHRKE, J. 2013. Asynchronous large-scale graph processing made easy. In *Conference on Innovative Data Systems Research (CIDR)*. Asilomar, California.

[59] WEISS, C., KARRAS, P., AND BERNSTEIN, A. 2008. Hexastore: Sextuple Indexing for Semantic Web Data Management. *Proceedings of the VLDB Endowment 1,* 1, 1008–1019.

[60] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. 2010. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. HotCloud'10. USENIX Association, Berkeley, CA, USA, 10–10.

[61] ZENG, K., YANG, J., WANG, H., SHAO, B., AND WANG, Z. 2013. A distributed graph engine for web scale rdf data. *Proc. VLDB Endow. 6,* 4, 265–276.

[62] ZHANG, Y., GAO, Q., GAO, L., AND WANG, C. 2011a. imapreduce: A distributed computing framework for iterative computation. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*. IPDPSW '11. IEEE Computer Society, Washington, DC, USA, 1112–1121.

[63] ZHANG, Y., GAO, Q., GAO, L., AND WANG, C. 2011b. Priter: a distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. SOCC '11. ACM, New York, NY, USA, 13:1–13:14.

[64] ZHU, X. AND GHAHRAMANI, Z. 2002. Learning from labeled and unlabeled data with label propagation. Tech. Rep. CMU-CALD-02-107, Carnegie Mellon University. June.